TECNICATURA UNIVERSITARIA EN INTELIGENCIA ARTIFICIAL (TUIA)

PROCESAMIENTO DE IMÁGENES (PDI)

Docentes de la cátedra:

Dr., Ing. Gonzalo Sad

Ing. Julian Álvarez

Trabajo Práctico 2 (TP2)

Estudiantes (GRUPO 7):

María Celi

Francisco J. Alomar

Bruno Longo

AÑO 2024 (1er CUATRIMESTRE), Rosario.

INTRODUCCIÓN

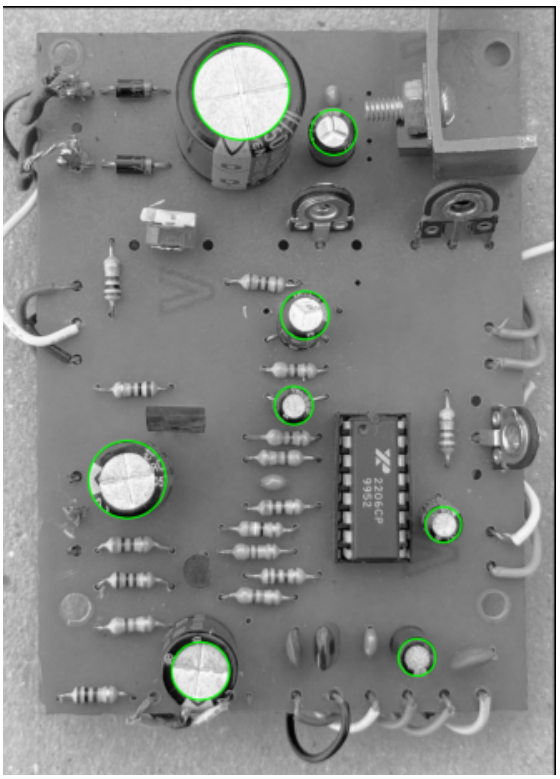
En el siguiente informe presentamos el procesamiento de imágenes en 2 ejercicios propuestos en el TP2. Las consignas constan de dos problemas a solucionar. El primero de ellos consiste en identificar componentes de un circuito eléctrico (resistencias, capacitores y chip) a partir de una imagen. El segundo, en reconocer las patentes de 12 autos distintos conjuntamente con los números y letras que las componen. Para ambos problemas optamos por soluciones diversas que explicitamos en el desarrollo. Además, hacemos mención de algunas dificultades que advertimos durante la resolución del trabajo práctico. Finalmente, esbozamos algunas conclusiones.

PROBLEMA 1 - Detección y clasificación de componentes electrónicos

a) Procesar la imagen

El algoritmo procede a identificar los capacitores, resistencias y chip de la placa de circuito. Comenzamos por los capacitores:

- Se carga la imágen a color y luego en escala de grises para simplificar el procesamiento.
- Posteriormente definimos un kernel de 3*3 con la función getStructuringElement() para aplicar la operación morfológica de dilatación con la función dilate(). Si bien podríamos prescindir de la dilatación para encontrar los capacitores (que son identificados como círculos), la operación los ajusta un poco mejor, lo cual sirve luego para su clasificación.
- Después, con la función HoughCircles() se detectan los círculos utilizando la transformada de Hough. Con el último *for*, por cada círculo de los círculos encontrados en base a la identificación de centro y radio, se dibujan los círculos sobre una copia de la imágen original en escala de grises, obteniendo la siguiente imágen:

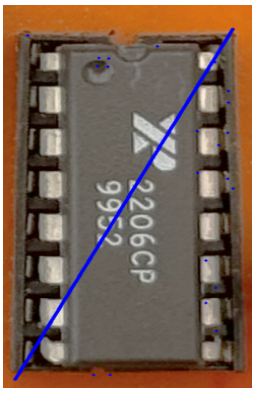


Para la detección del chip implementamos la siguiente secuencia de pasos:

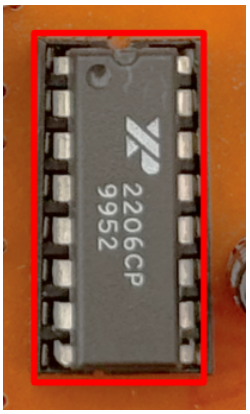
- A la imagen original en escala de grises la binarizamos y aplicamos una secuencia de dilataciones y erosiones con las funciones dilate() y erode(), respectivamente. Luego hacemos un inversión de la binarización, dilatamos, obtenemos bordes con la función Canny() y volvemos a dilatar. Si no se aplica este proceso, la imagen binarizada invertida es la siguiente:

- Por el contrario, si aplicamos el proceso descrito, los contornos de las figuras se destacan notablemente, facilitando la detección del chip. En la imágen binarizada invertida después de aplicar las dilataciones, erosiones y la detección de borde de Canny, se destaca de modo más considerable:

- Como puede apreciarse, el chip aún mantiene un perímetro abierto en su parte superior que complejiza segmentar su contorno. Como solución, optamos por tomar su diagonal, hallada mediante la simplificación del polígono aproximado por cv2.approxPolyDP():



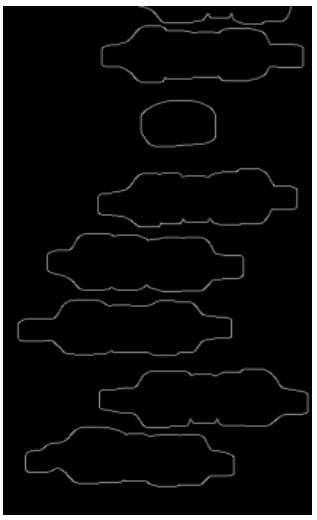
- Gracias a esto, podemos dibujar el rectángulo, ya que disponemos del ancho y del alto como relación trigonométrica o como diferencia entre las coordenadas x e y de los extremos.



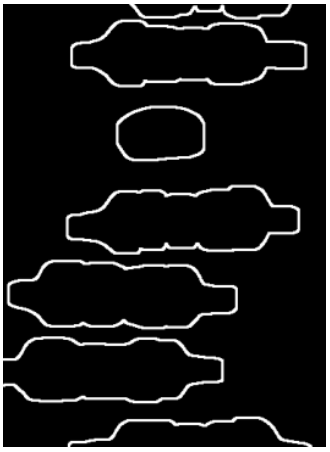
Para la detección de las resistencias implementamos la siguiente secuencia de pasos:

- Primero, preprocesamos la imagen a través de filtros GaussianBlur() con kernel cuadrado de 25x25, un umbralado para valores de gris mayores a 140 y un dilate() con un kernel cuadrado de 11x11. Con esta primera etapa, realizamos la primera

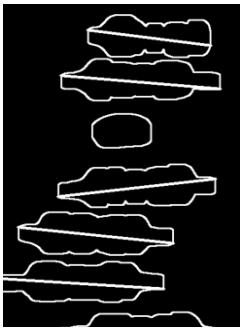
detección de bordes con Canny(), que logra un contorno satisfactorio aunque “demasiado fino” en sectores:



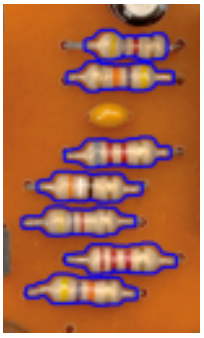
Por lo que asistidos nuevamente por la máscara dilate() logramos contorno completos en todas las resistencias que pudimos notar:



- Finalmente, para segmentar solamente a las resistencias entre los demás componentes, optamos por utilizar su (similar al *chip*) “diagonal”. Esta diagonal es el polígono aproximado con una simplificación específica para que podamos segmentar sólo las resistencias. Cabe destacar que antes de ello, intentamos servirnos del perímetro, del área y de la cantidad de lados (además de combinaciones entre ellos) sin encontrar la solución buscada.



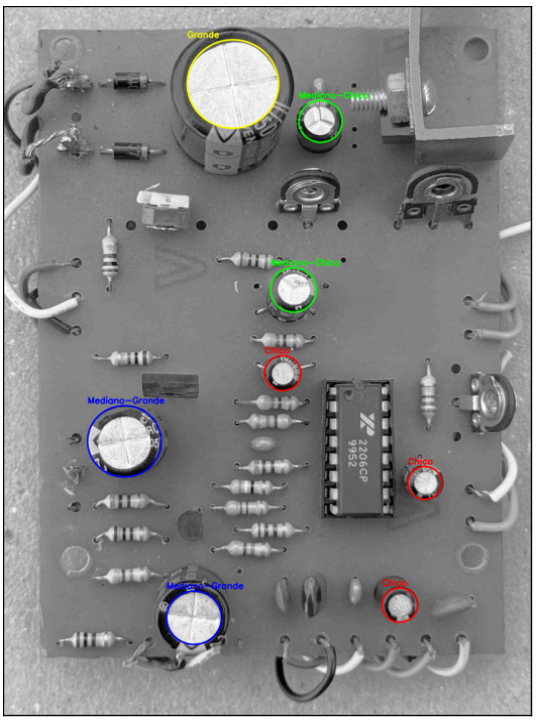
- El área es definida solamente para aquellos contornos que tengan una diagonal entre 205 y 295 píxeles, lo que fue especificidad suficiente para segmentar (aunque no para contar, como se ve en el próximo punto).



b) Clasificar capacitores electrolíticos

A continuación, se describen los pasos tomados para implementar el algoritmo que segmenta y clasifica capacitores electrolíticos en una imagen, agrupándolos según su tamaño y generando una imagen de salida con esta clasificación:

- Se reutiliza el código de segmentación de capacitores del punto “a” para luego clasificarlos.
- Se utiliza la transformada de Hough para identificar círculos en la imagen dilatada, lo que permite detectar los capacitores electrolíticos basados en su forma circular. Los parámetros de la función HoughCircles se ajustan para detectar capacitores de diferentes tamaños.
- Los capacitores detectados se clasifican en cuatro categorías basadas en el radio de los círculos detectados:
    - Chico: radio menor a 80.
    - Mediano-Chico: radio entre 81 y 110.
    - Mediano-Grande: radio entre 111 y 160.
    - Grande: radio mayor a 160.
- Se cuenta la cantidad de capacitores en cada categoría y se dibujan círculos de colores en la imagen original para cada capacitor, con un color específico asignado a cada categoría. Se añade una etiqueta de clasificación junto a cada capacitor.



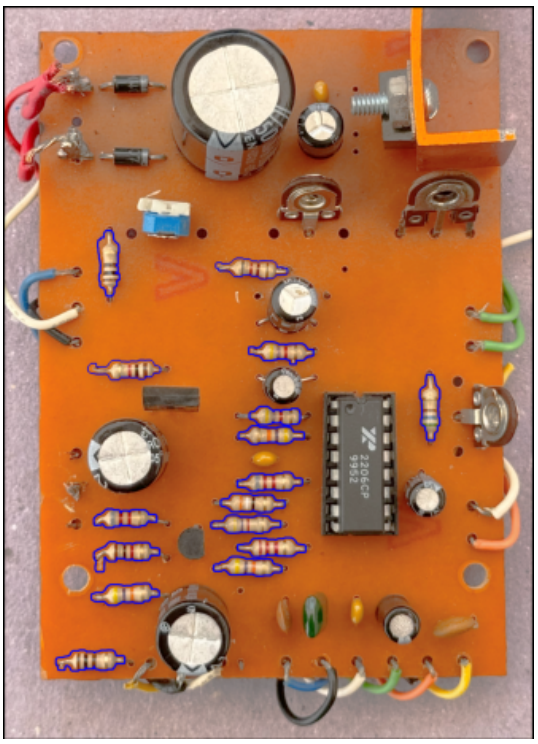
- Se genera una nueva imagen de salida que muestra los capacitores clasificados por tamaño, cada uno con su círculo y etiqueta correspondiente.
- Se muestra la imagen resultante con la clasificación de capacitores y se imprime el conteo de capacitores por cada categoría.

```
Chico: 3
Mediano-Chico: 2
Mediano-Grande: 2
Grande: 1
```

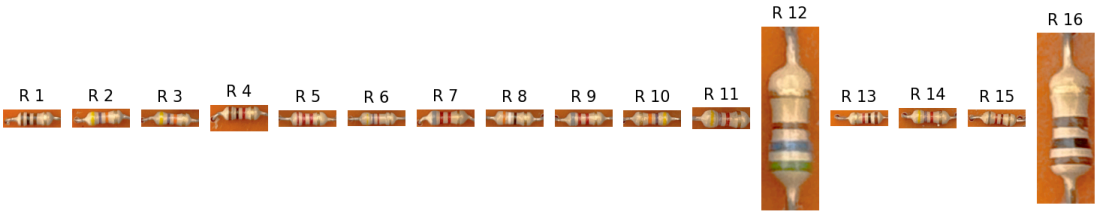
c) Contar resistencias eléctricas

Al igual que el punto anterior, describimos una implementación que segmenta objetos, en este caso, resistencias para para determinar su cantidad:

- Se reutiliza el código de segmentación de resistencias eléctricas del punto “a” para encontrar los contornos en la imagen procesada que representan las posibles resistencias.



- Se inicializa un diccionario para almacenar las resistencias detectadas y una lista para guardar los puntos procesados.
- Para cada contorno detectado, se calcula su perímetro y su aproximación a un polígono. Luego, se evalúan las diagonales del polígono:
    - Si la longitud de una diagonal se encuentra dentro del rango típico de las resistencias (205-295 píxeles) y la resistencia no se ha contado previamente (verificado por la lista de puntos), se considera una resistencia válida.
    - Se dibuja el contorno de la resistencia detectada en una imagen de salida y se agrega al diccionario de resistencias.
- Se genera una imagen de salida que muestra las resistencias detectadas.
- Se muestra cada resistencia detectada en subplots separados:



- Se imprime el número de resistencias detectadas en la consola.

En este punto tuvimos una objetivación frente al Problema y Solución Relacionados con la Detección de Contornos Doble:

- Problema: El método de detección de contornos basado en la detección de bordes y la aproximación de polígonos puede detectar tanto los bordes exteriores como interiores de componentes cilíndricos, como las resistencias. Esto resulta en la detección de contornos dobles, que pueden conducir a una sobreestimación en el conteo de resistencias.
- Solución: Para resolver este problema, se agregó una condición que verifica si el punto inicial del contorno actual es similar al del contorno anterior. Si es así, el contorno no se cuenta, evitando la doble contabilización. Este ajuste mejora la precisión del conteo.

```
if 205 < long_diagonal < 295 and all(abs(pt1 - puntos[-1]) > np.array([10, 10])):
    puntos.append(pt1)
    resistencias[j] = long_diagonal
    cv2.drawContours(output_image_res, [area], -1, (0, 0, 255), 3)
```

La condición agregada asegura que no se cuentan dos contornos muy similares, mejorando la precisión en el conteo de resistencias al evitar la doble contabilización de bordes interiores y exteriores.

PROBLEMA 2 - Encontrar números y letras de patentes.

Comenzamos por describir el enfoque general que motivó el diseño del algoritmo. Luego, explicamos la secuencia de pasos con una de las patentes para que se comprenda el resultado final. Básicamente, el código hace dos *crops* sobre cada imagen original. Ambos recortes tienen el fin de generar menos datos y más información para el procesamiento. Cuando analizamos la totalidad de las imágenes advertimos que el punto de toma de la cámara era el de una persona apuntando de arriba hacia abajo. Esto permitió suponer un primer recorte que funcione para todas ellas, es decir, sustraer secciones de las fotos donde no encontraríamos patentes (por ejemplo, en las esquinas superiores izquierdas). Utilizamos estos primeros *crops* para encontrar los componentes conectados que conforman los números y letras de las respectivas patentes. El problema fue que por el tamaño de la imágen, si bien el algoritmo encontraba la mayoría de caracteres, también identificaba componentes conectados que resultaban “ruido” para el análisis, lo que hacía

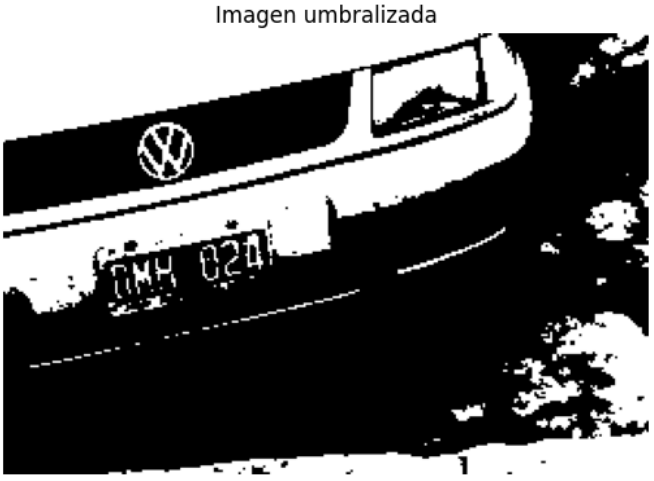
defectuosa la correcta identificación. Sin embargo, advertimos que el *gap* entre el conjunto de letras y de números para cada patente tenía una variación de 8 a 14 píxeles según la imágen. Por lo tanto, en esta primera instancia bastaba sólo encontrar el *gap* y tomar sus coordenadas como centro de un nuevo *crop*. De este modo, nos ahorramos encontrar todas las componentes de la patente entre tanto “ruido”. Una vez hecho el segundo *crop*, se aprecia que gran porcentaje de la imágen resultante es la patente que se quiere hallar. Sobre ella repetimos una implementación similar a la de ubicar el *gap*, sólo que el objetivo es encontrar la cantidad exacta de números y letras de cada patente, aplicando un tratamiento de la imágen que facilita su identificación.

Ahora pasamos a explicar el algoritmo mostrando resultados parciales para una mayor claridad en base al archivo img03.png. Aplicamos un primer recorte (el mismo para todas las imágenes). Pasamos la imágen a escala de gris y como resultado obtenemos:

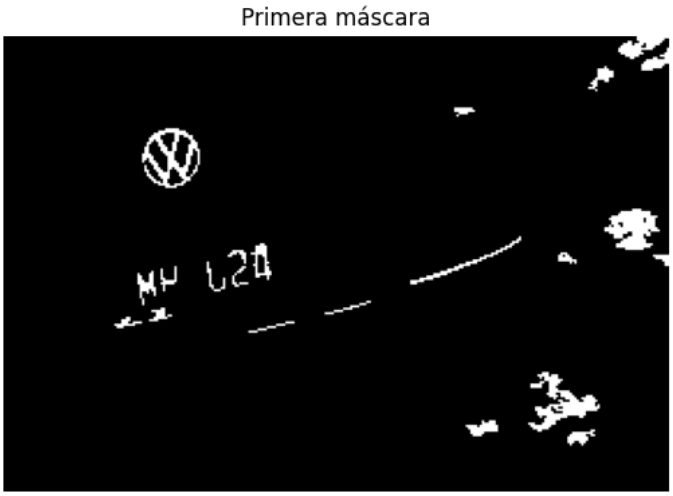
Primer crop

Para la imágen hallamos la mediana de la escala de grises. Con este valor más una constante se ingresa a un ciclo *while* que se detiene si al ir decrementando la mediana encuentra el *gap* antes mencionado. Al ingresar al *while* se siguen los siguientes pasos. Las imágenes que mostramos son las que se obtienen cuando encuentra el *gap* en el caso particular de la img03.png:

- Se declara umbral_inferior (mediana + constante) la cual es utilizada para binarizar la imágen en la función cv2.threshold(), generando la siguiente variable imagen_umbral:



- El resultado de la función anterior es un parámetro de la función cv2.connectedComponentsWithStats(). En ella, desempaquetamos dos variables: etiquetas y estadisticas, de las cuales, la primera es una matriz del mismo tamaño que la imagen de entrada, donde cada elemento tiene un valor que representa la etiqueta del componente conectado al que pertenece, y la segunda es la matriz Stats.
- Se definen los umbrales de área para identificar componentes de tamaño plausible que contengan el tamaño de una placa en píxeles (umbral_area = 17, umbral_area2 = 300).
- Se crea una mascara_componentes = np.zeros_like(imagen_umbral), que es una matriz del mismo tamaño que imagen_umbral, inicializada en cero (todo negro).
- El bucle for itera sobre cada componente (descartando el fondo que se etiqueta como 0). Por cada componente se extrae su área de la matriz de estadisticas (que es la matriz Stats). Con ello, se obtiene la siguiente matriz mascara_componentes:



- Se vuelve a utilizar la función cv2.connectedComponentsWithStats() a la cual se le pasa como parámetro la matriz anterior. Con las variables que se desempaquetan de la función se genera otro ciclo for que busca componentes conectados por aspecto, alto y ancho aproximados en que coincidan los caracteres de la patente. Y la matriz que se obtiene es la siguiente:

- Una vez obtenida la matriz, se genera la stats_ordenadas que se ordena de acuerdo la componente x:

[[0 0 288 198 56887]
[135 120 7 14 31]
[144 120 6 12 23]

[153 118 6 12 30]
[170 115 6 13 18]
[187 112 6 13 35]]

- Y también el algoritmo retorna una lista con la distancia exceptuando el fondo. En este caso son cuatro distancias:

  [2, 3, 11, 11]

- Se recorre la lista y dado que encuentra un valor entre 8 y 14, el *gap* corresponde al centro de la patente. Con la información recolectada hacemos el segundo *corp*, en el cual, gran porcentaje de la imágen es la placa patente y sus componente:

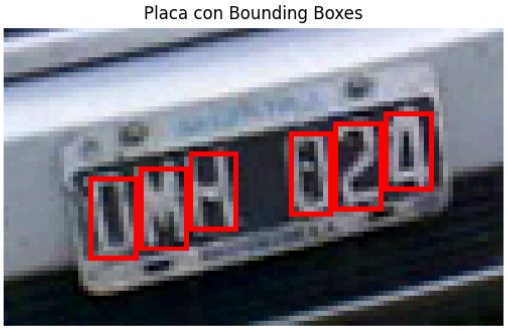
Es así que se genera una lista de crop_patentes cercanas a las placas para cada auto. Tratamos las imágen con el fin de mejorar la nitidez mediante las funciones cv2.GaussianBlur() y cv2.addWeighted(). Además, convertimos la imagen a escala de grises y utilizamos la función cv2.createCLAHE(), que es un modo práctico de aplicar una ecualización local de histograma que deviene en un contraste más adecuado para la resolución del problema. El fin del procesamiento es lograr un mejor umbralado. Luego, el proceso es similar al anterior, sólo que al estar más cerca las patentes, eliminamos componentes que interferían para encontrar aquellos deseados. Además, variamos los parámetros de área y umbrales que cambian con la cercanía de la imágen. El ciclo *while* se controla hasta hallar 7 componentes conectados que, menos el fondo, son los 6 caracteres de las patentes. A continuación, mostramos las imágenes con un tratamiento similar al anterior:

- Al segundo crop lo pasamos a escala de grises y realizamos umbralización de acuerdo a la mediana:

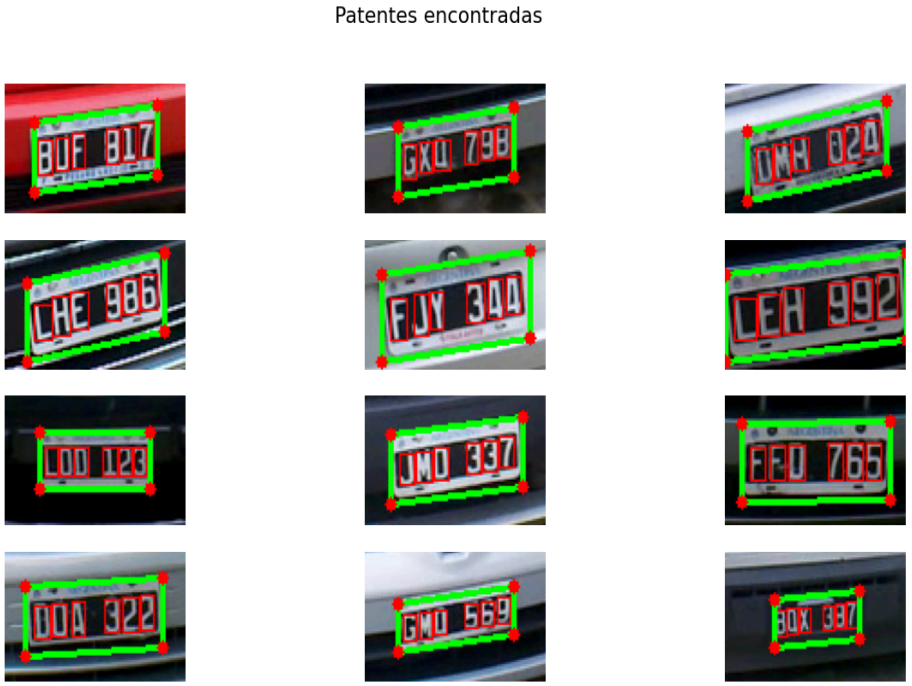
- Luego, generamos una máscara con áreas de componentes conectadas que puedan coincidir con algún caracter de la patente:

- Posteriormente, generamos otra máscara que filtra la anterior por aspecto (alto y ancho) de sus componentes conectados, también de acuerdo a número y letras:

- Con la matriz Stats ordenada determinamos su área y así los *bounding boxes* para cada componente y los ploteamos sobre el segundo crop a color:



Para encuadrar las patentes optamos por la siguiente resolución. Dado que existe una relación entre los componentes conectados con caracteres, establecimos un ancho y alto promedio de todos ellos por patente. Así, generamos cuatro puntos que se desplazan hacia arriba y hacia abajo tomando como referencia el primer y último componente conectado, la mitad del ancho y alto promedio, multiplicado por una constante. Esto determina cuatro vértices con los que dibujamos un paralelogramo que contiene aproximadamente a las patentes. Finalmente, mostramos el resultado que se devuelve al ejecutar el algoritmo:



CONCLUSIÓN

El proceso de trabajo ha demostrado que frente a los problemas hay diversas resoluciones posibles. A medida que avanzamos en ellas, advertimos que tal vez podríamos haberlas optimizado, pero en general, fuimos guiados por el objetivo de explorar alternativas más que hacerlo de la mejor manera. Por ejemplo, al resolver el segundo problema, las dificultades encontradas al principio hicieron que optamos por “acercarnos” lo más posible a la imagen a tratar, aunque esto redundara (como se denota en el código) en aplicar un tratamiento similar dos veces. Una vez concluido el proceso, advertimos que tal vez podríamos haber ahorrado un paso y trabajar con un solo ciclo while, sobre la imagen original, o, también, hallar las patentes no por una proporción de acuerdo a los números y letras, sino también con el mismo enfoque que encontramos los caracteres.

Además, en el caso del conteo de resistencias, encontramos que el método de detección de contornos que utilizamos tiende a detectar tanto los bordes exteriores como los interiores de componentes cilíndricos, lo que resultó en la detección de contornos dobles. Para abordar este problema, implementamos una condición adicional que verifica si el punto inicial del contorno actual es muy similar al del contorno anterior, evitando así la doble contabilización de contornos similares.

En síntesis, queremos decir que a los fines del TP2 obtuvimos soluciones eficaces más que eficientes que resultaron en un proceso de aprendizaje profundo sobre el procesamiento de imágenes y la importancia de la optimización de algoritmos. Este aprendizaje nos permitirá abordar problemas futuros con una mejor comprensión y un enfoque más refinado.