Regiones en imágenes binarias

Temas

- Encontrar regiones conectadas en una imagen
- Encontrar contornos de regiones
- Representación de regiones en una imagen
- Propiedades de las imágenes binarias

Introducción

- En imágenes binarias los pixeles solo pueden tomar dos valores.
- Estos valores representan el primer plano (foreground) y el fondo (background).
- El objetivo de esta parte es definir algoritmos para encontrar el número y el tipo de objetos en una imagen binaria.

Introducción

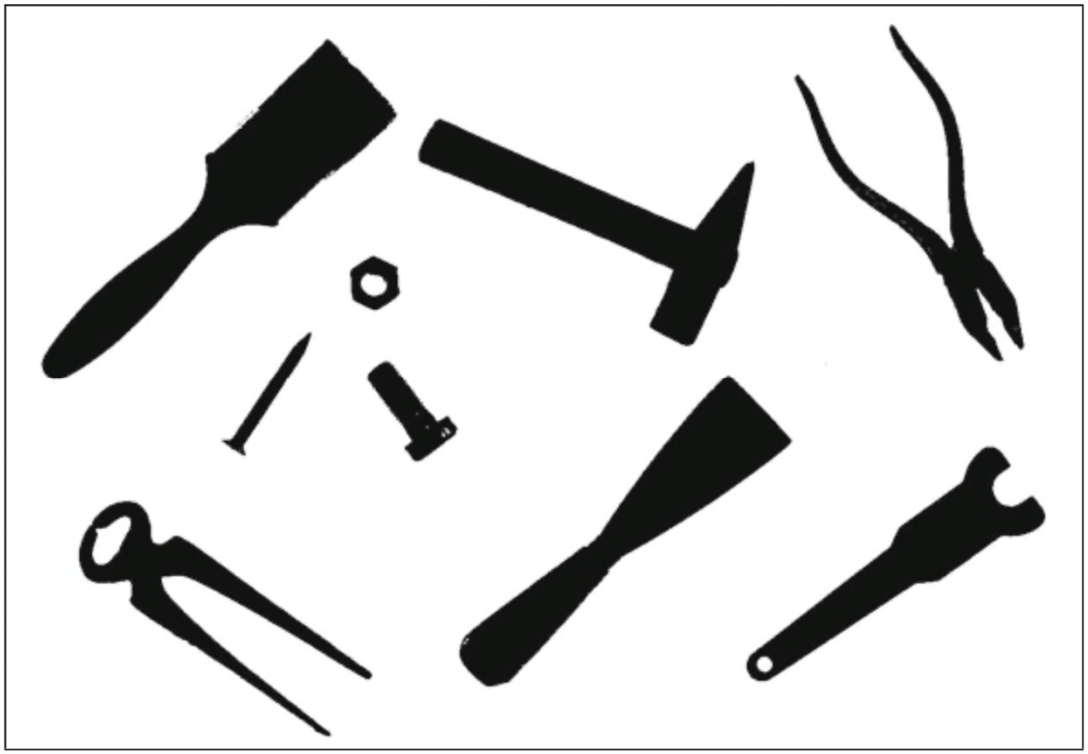


Fig. 10.1
Binary image with nine components. Each component corresponds to a connected region of (black) foreground pixels.

Búsqueda de regiones binarias

- Las tareas más importantes son:

a) Qué pixeles corresponden a que regiones.
b) Cuántas regiones hay en la imagen.
c) Dónde están localizadas estas regiones.

- En general, el proceso de búsqueda analiza pixeles vecinos para construir regiones contiguas.
- A los pixeles de cada región se les asigna un número único o etiqueta.

Algoritmos de búsqueda de regiones

1. Inundación (flood filling). La región se llena en todas direcciones a partir de un pixel (semilla) dentro de la región.
2. Marcado secuencial de regiones. Se recorre la imagen de arriba abajo marcando las regiones conforme se encuentran.

Convenciones

- Hay que especificar que definición de vecindad se va a usar: 4-vecinos u 8-vecinos.
- Los valores posibles de los pixeles en la imagen $I(u,v)$ son:
- $$I(u,v) = \begin{cases} 0 & \text{fondo} \\ 1 & \text{primer plano} \\ 2, 3, \ldots & \text{etiqueta de región} \end{cases}$$

Inundación

- Consiste en buscar un pixel del primer plano sin marcar y llenar (visitar y marcar) el resto de los pixeles vecinos en la región.
- Se le llama inundación porque es como si un flujo de agua saliera del pixel del comienzo e inundara un terreno plano.
- Hay 3 formas de implementar este algoritmo:

a) Recursivo.
b) Primero en profundidad (depth-first).
c) Primero en anchura (breadth-first).

Inundación

```
1:  RegionLabeling(I)
        Input: I, an integer-valued image with initial values 0 = background, 1 = foreground. Returns nothing but modifies the image I.
2:      label ← 2                          ▷ value of the next label to be assigned
3:      for all image coordinates u, v do
4:          if I(u, v) = 1 then            ▷ a foreground pixel
5:              FloodFill(I, u, v, label)  ▷ any of the 3 versions below
6:              label ← label + 1.
7:      return
```

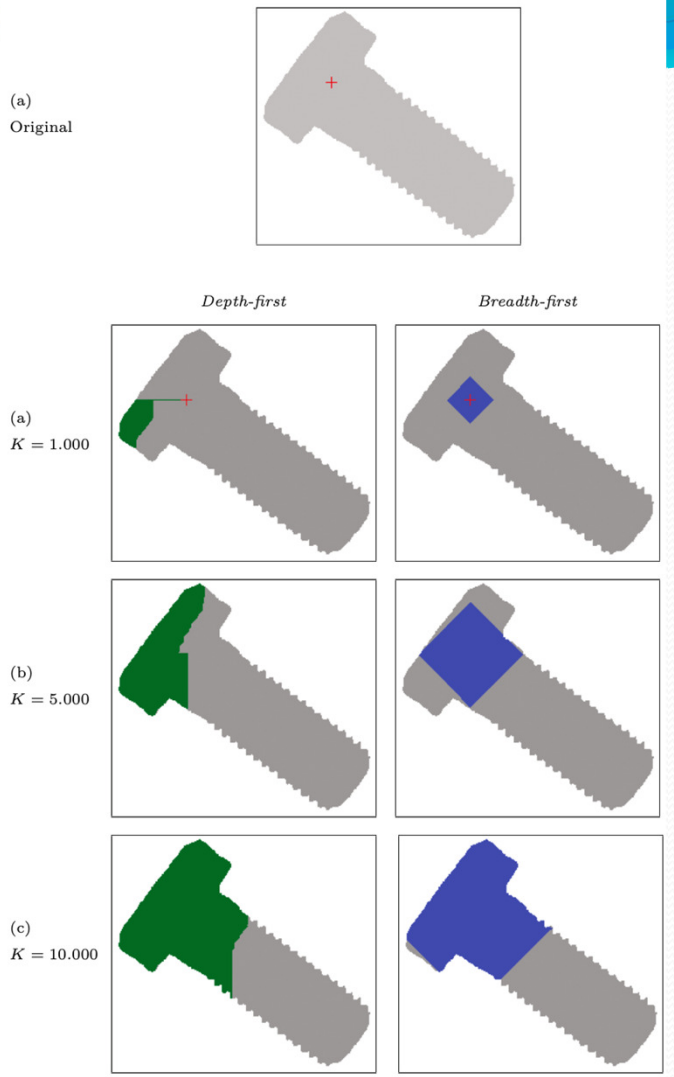
```
8:  FloodFill(I, u, v, label)              ▷ Recursive Version
9:      if u, v is within the image boundaries and I(u, v) = 1 then
10:         I(u, v) ← label
11:         FloodFill(I, u+1, v, label)    ▷ recursive call to FloodFill()
12:         FloodFill(I, u, v+1, label)
13:         FloodFill(I, u, v−1, label)
14:         FloodFill(I, u−1, v, label)
15:     return
```

Inundación

```
16: FloodFill(I, u, v, label)                          ▷ Depth-First Version
17:     S ← ( )                                        ▷ create an empty stack S
18:     S ← (u, v) ⌣ S                                 ▷ push seed coordinate (u, v) onto S
19:     while S ≠ ( ) do                               ▷ while S is not empty
20:         (x, y) ← GetFirst(S)
21:         S ← Delete((x, y), S)                      ▷ pop first coordinate off the stack
22:         if x, y is within the image boundaries and I(x, y) = 1 then
23:             I(x, y) ← label
24:             S ← (x+1, y) ⌣ S                       ▷ push (x+1, y) onto S
25:             S ← (x, y+1) ⌣ S                       ▷ push (x, y+1) onto S
26:             S ← (x, y−1) ⌣ S                       ▷ push (x, y−1) onto S
27:             S ← (x−1, y) ⌣ S                       ▷ push (x−1, y) onto S
28:     return
29: FloodFill(I, u, v, label)                          ▷ Breadth-First Version
30:     Q ← ( )                                        ▷ create an empty queue Q
31:     Q ← Q ⌣ (u, v)                                 ▷ append seed coordinate (u, v) to Q
32:     while Q ≠ ( ) do                               ▷ while Q is not empty
33:         (x, y) ← GetFirst(Q)
34:         Q ← Delete((x, y), Q)                      ▷ dequeue first coordinate
35:         if x, y is within the image boundaries and I(x, y) = 1 then
36:             I(x, y) ← label
37:             Q ← Q ⌣ (x+1, y)                       ▷ append (x+1, y) to Q
38:             Q ← Q ⌣ (x, y+1)                       ▷ append (x, y+1) to Q
39:             Q ← Q ⌣ (x, y−1)                       ▷ append (x, y−1) to Q
40:             Q ← Q ⌣ (x−1, y)                       ▷ append (x−1, y) to Q
41:     return
```

Inundación iterativ

Fig. 10.2
Iterative *flood filling*—comparison between the *depth-first* and *breadth-first* approach. The starting point, marked + in the top two image (a), was arbitrarily chosen. Intermediate results of the *flood fill* process after 1000 (a), 5000 (b), and 10,000 (c) marked pixels are shown. The image size is 250 × 242 pixels.



Implementación en Java

- La versión recursiva del algoritmo no es práctica.
- Un runtime normal de Java no soporta más de 10,000 llamadas recursivas al método FloodFill sin que se acabe la memoria de pila.
- Esto solo basta para imágenes de menos de 200 x 200 pixeles.
- La versión primero en profundidad se puede implementar usando una pila.
- La versión primero en anchura se puede implementar usando una cola.

Implementación en Java

Depth-first version (using a *stack*):

```
1  void floodFill(int u, int v, int label) {
2    Deque<Point> S = new LinkedList<Point>(); // stack S
3    S.push(new Point(u, v));
4    while (!S.isEmpty()) {
5      Point p = S.pop();
6      int x = p.x;
7      int y = p.y;
8      if ((x >= 0) && (x < width) && (y >= 0) && (y < height)
9          && ip.getPixel(x, y) == 1) {
10       ip.putPixel(x, y, label);
11       S.push(new Point(x + 1, y));
12       S.push(new Point(x, y + 1));
13       S.push(new Point(x, y - 1));
14       S.push(new Point(x - 1, y));
15     }
16   }
17 }
```

Implementación en Java

Breadth-first version (using a *queue*):

```
18 void floodFill(int u, int v, int label) {
19   Queue<Point> Q = new LinkedList<Point>(); // queue Q
20   Q.add(new Point(u, v));
21   while (!Q.isEmpty()) {
22     Point p = Q.remove(); // get the next point to process
23     int x = p.x;
24     int y = p.y;
25     if ((x >= 0) && (x < width) && (y >= 0) && (y < height)
26         && ip.getPixel(x, y) == 1) {
27       ip.putPixel(x, y, label);
28       Q.add(new Point(x + 1, y));
29       Q.add(new Point(x, y + 1));
30       Q.add(new Point(x, y - 1));
31       Q.add(new Point(x - 1, y));
32     }
33   }
34 }
```

Marcado secuencial de regiones

- También conocido como *etiquetado de regiones*.
- Consiste en dos pasos:

1. Etiquetar de forma preliminar las regiones de la imagen.
2. Resolver colisiones de etiquetas (si los pixeles de una misma región tiene más de una etiqueta).

Paso 1

- Etiquetado inicial.
- La imagen se recorre de arriba abajo y de izquierda a derecha asignándole a cada pixel una etiqueta.
- La definición de vecindad que se use determina que vecinos se examinan.

$$\mathcal{N}_4 = \begin{array}{ccc} & N_1 & \\ N_2 & \times & N_0 \\ & N_3 & \end{array} \quad \text{or} \quad \mathcal{N}_8 = \begin{array}{ccc} N_3 & N_2 & N_1 \\ N_4 & \times & N_0 \\ N_5 & N_6 & N_7 \end{array}. \qquad (10.2)$$

Paso 1

- Para cada pixel:

a) Si ningún vecino tiene etiqueta, entonces al pixel se le asigna una nueva etiqueta.

b) Si exactamente un vecino tiene etiqueta, entonces esa etiqueta se le asigna al pixel.

c) Si más de un vecino tiene etiqueta (pueden ser distintas), entonces alguna de esas etiquetas se le asigna al pixel. Si las etiquetas de los vecinos son distintas se registra la colisión.

Paso 1

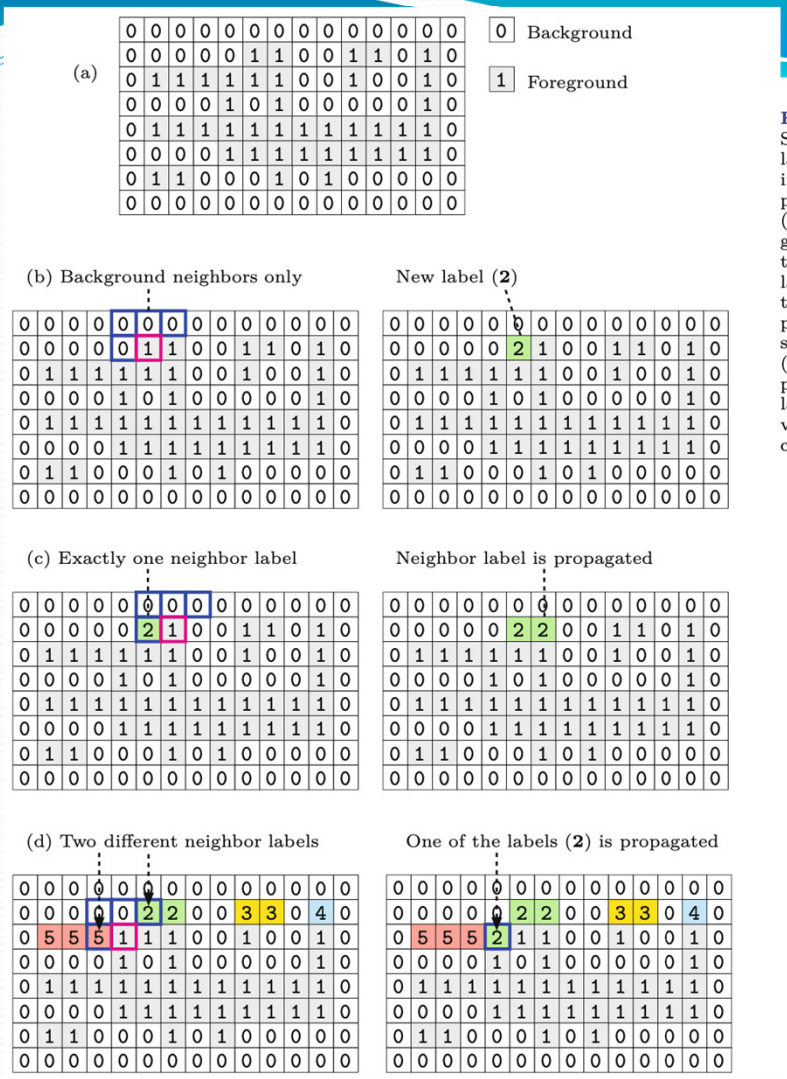


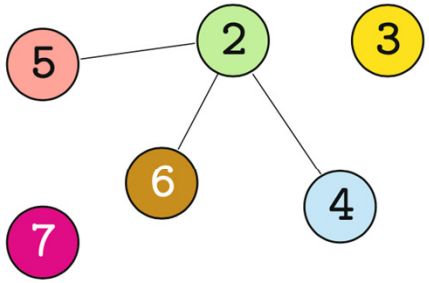
Fig. 10.3
Sequential region labeling—label propagation. Original image (a). The first foreground pixel (marked **1**) is found in (b): all neighbors are background pixels (marked **0**), and the pixel is assigned the first label (**2**). In the next step (c), there is exactly *one* neighbor pixel marked with the label **2**, so this value is propagated. In (d) there are *two* neighboring pixels, and they have differing labels (**2** and **5**); one of these values is propagated, and the collision (**2**, **5**) is registered.

Paso 1

- Al final del paso 1 todos los pixeles han sido etiquetados y todas las colisiones han sido registradas.

0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	2	2	0	0	3	3	0	4	0
0	5	5	5	2	2	2	0	0	3	0	0	4	0
0	0	0	0	2	0	2	0	0	0	0	0	4	0
0	6	6	2	2	2	2	2	2	2	2	2	2	0
0	0	0	0	2	2	2	2	2	2	2	2	2	0
0	7	7	0	0	0	2	0	2	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0

(a)



(b)

Fig. 10.4
Sequential region labeling—intermediate result after step 1. Label collisions indicated by circles (a); the nodes of the undirected graph (b) correspond to the labels, and its edges correspond to the collisions.

Paso 2

- Solución de colisiones de etiquetas.
- Las etiquetas de todos los pixeles se actualizan para que tengan las misma etiqueta.
- Una opción es seleccionar la etiqueta con el valor mínimo.

Paso 2

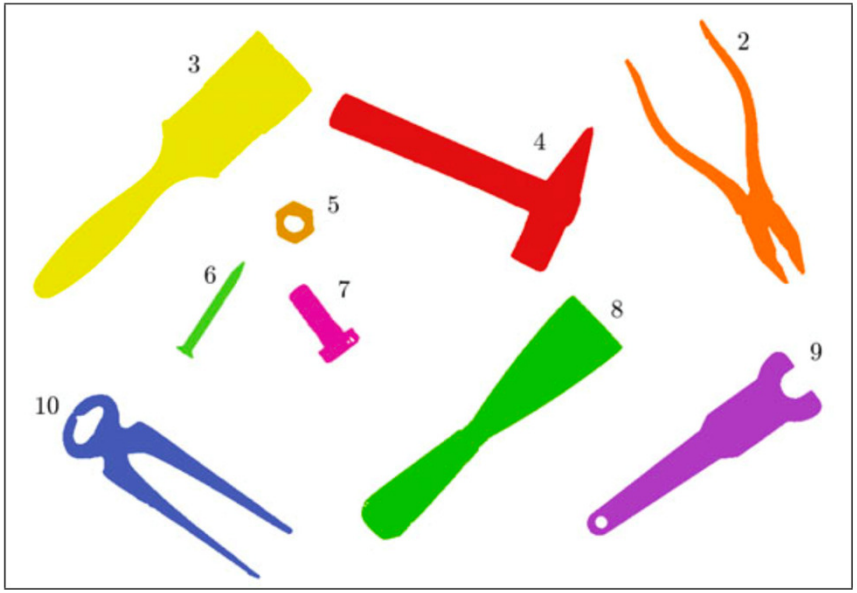
- Resultado final después del paso 2:

Fig. 10.5
Sequential region labeling—final result after step 2. All equivalent labels have been replaced by the smallest label within that region.

0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	2	2	0	0	3	3	0	2	0
0	2	2	2	2	2	2	0	0	3	0	0	2	0
0	0	0	0	2	0	2	0	0	0	0	0	2	0
0	2	2	2	2	2	2	2	2	2	2	2	2	0
0	0	0	0	2	2	2	2	2	2	2	2	2	0
0	7	7	0	0	0	2	0	2	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0

Ejemplo

Fig. 10.6
Example of a complete region labeling. The pixels within each region have been colored according to the consecutive label values $2, 3, \ldots, 10$ they were assigned. The corresponding region statistics are shown in the table (total image size is 1212×836).



Label	Area (*pixels*)	Bounding Box (*left, top, right, bottom*)	Centroid (x_c, y_c)
2	14978	(887, 21, 1144, 399)	(1049.7, 242.8)
3	36156	(40, 37, 438, 419)	(261.9, 209.5)
4	25904	(464, 126, 841, 382)	(680.6, 240.6)
5	2024	(387, 281, 442, 341)	(414.2, 310.6)
6	2293	(244, 367, 342, 506)	(294.4, 439.0)
7	4394	(406, 400, 507, 512)	(454.1, 457.3)
8	29777	(510, 416, 883, 765)	(704.9, 583.9)
9	20724	(833, 497, 1168, 759)	(1016.0, 624.1)
10	16566	(82, 558, 411, 821)	(208.7, 661.6)

Comparación

- Inundación vs marcado secuencial de regiones.
- Inundación recursiva solo es práctica para imágenes pequeñas.
- Marcado secuencial de regiones es un algoritmo complejo de implementar.
- No ofrece una clara ventaja sobre los algoritmos de inundación iterativos.
- En la práctica, la inundación primero en anchura es la opción para imágenes grandes y complejas.

Componentes conectados en OpenCV

- OpenCV proporciona cuatro funciones de análisis de componentes conectados:

1. cv.connectedComponents
2. cv.connectedComponentsWithStats
3. cv.connectedComponentsWithAlgorithm
4. cv.connectedComponentsWithStatsWithAlgorithm

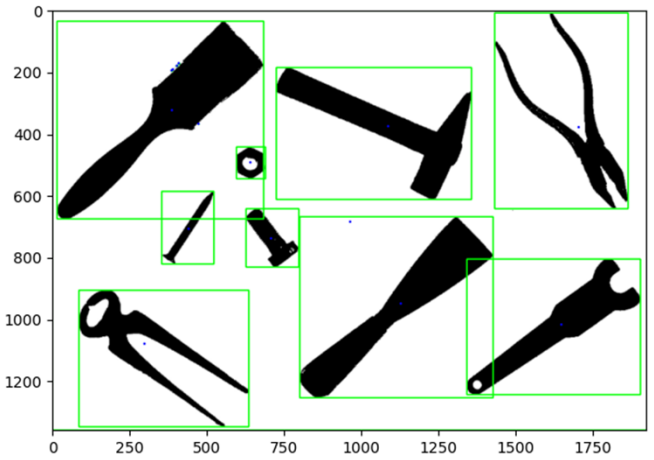
- En los siguientes ejemplos se usa cv.connectedComponentsWithStats.

Componentes conectados en OpenCV

- cv.connectedComponentsWithStats devuelve la siguiente información:
- El bounding box (cuadro delimitador) del componente conectado.
- El área en pixeles del componente.
- Las coordenadas del centroide (x, y) del componente.

Ejemplo

- Archivo: segmentation/connected_components_all.py.



- El programa reporta 15 regiones.

Ejemplo

- Archivo: connected_components_plate.py.

- El programa reporta 86 regiones.

Contornos

- El objetivo es encontrar una lista ordenada de los pixeles que forman el contorno de una región conectada (forma o figura).
- Cada región conectada en una imagen tiene un solo *contorno exterior*.
- Si tiene hoyos, la región puede tener varios *contornos internos*.
- Dentro de los hoyos pueden haber pequeñas regiones conectadas.

Contornos

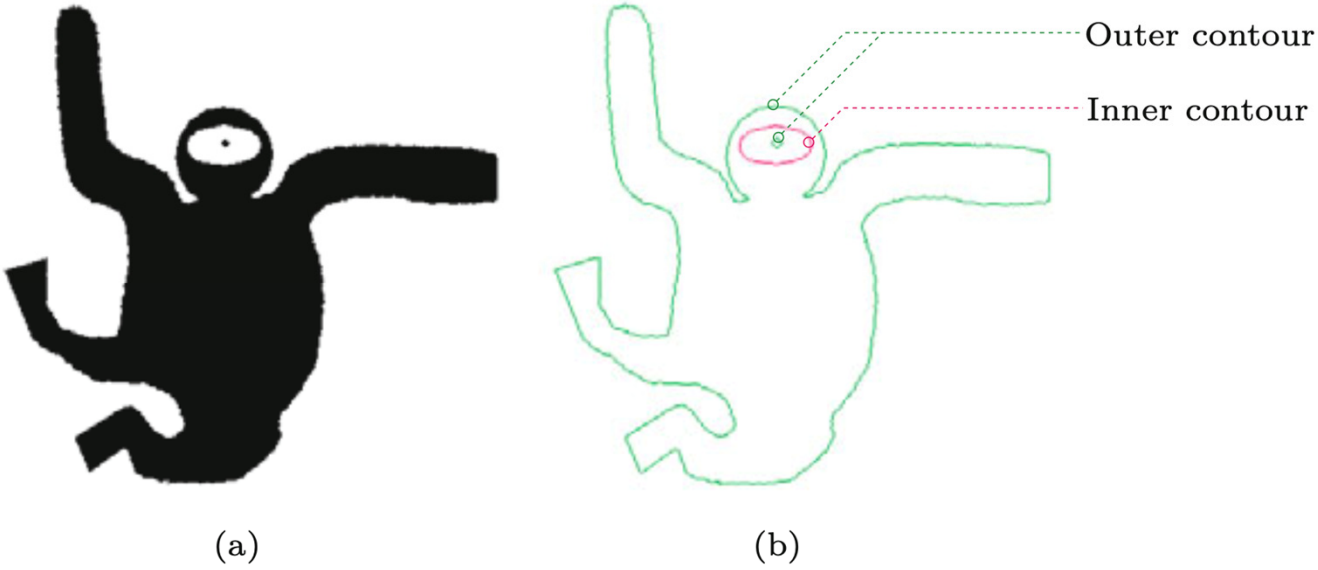


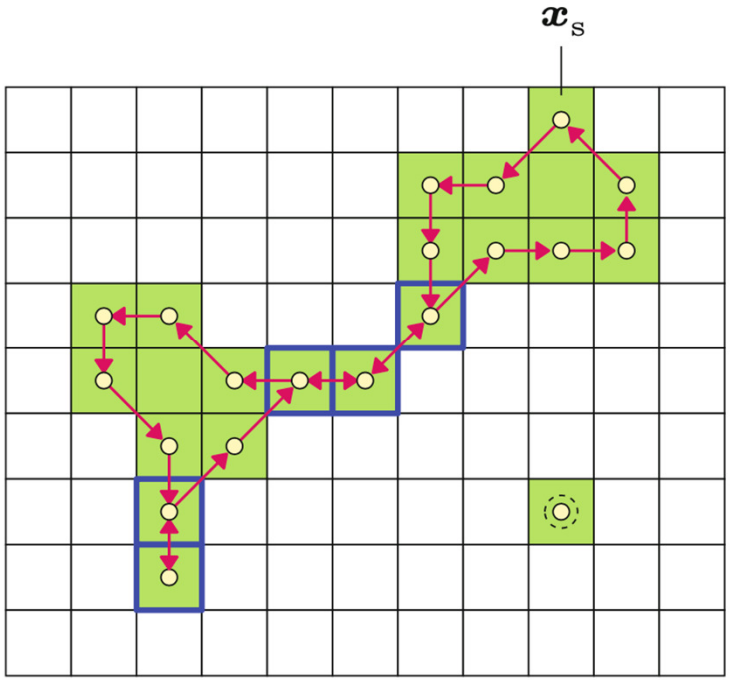
Fig. 10.7
Binary image with outer and inner contours. The outer contour lies along the outside of the foreground region (dark). The inner contour surrounds the space within the region, which may contain further regions (holes), and so on.

Contornos

- Otra complicación es cuando la conexión entre dos regiones tiene una anchura de 1 pixel.
- En este caso el contorno pasa por el mismo pixel varias veces y desde distintas direcciones.
- Esto implica que no basta con recorrer el contorno y regresar al pixel inicial para terminar el proceso de búsqueda del contorno.
- Hay que tomar en cuenta otros factores, como la dirección de recorrido.

Contornos

Fig. 10.8
The path along a contour as an ordered sequence of pixel coordinates with a given start point $\boldsymbol{x}_s$. Individual pixels may occur (be visited) more than once within the path, and a region consisting of a single isolated pixel will also have a contour (bottom right).



Contornos

- Una forma de encontrar el contorno es mediante *trazado de contornos*:

1. Encontrar las regiones conectadas usando, por ejemplo, inundación iterativa.
2. Comenzando con un pixel en el contorno, proceder alrededor de cada región.

- De la misma forma se puede encontrar un contorno interior comenzando en un pixel en la frontera de un hoyo.

Algoritmo

- Detalles en Burger & Burge, *Digital Image Processing*, p. 220

Algoritmo

1. La imagen se recorre de arriba hacia abajo y de izquierda a derecha.
2. En cada pixel hay 3 casos:

a) Hay una transición de un pixel del fondo a un pixel sin marcar del frente.

Esto indica que el pixel pertenece al contorno exterior de una nueva región.

Se asigna una nueva etiqueta.

Se recorre y marca el contorno.

Todos los pixeles del fondo vecinos a la región se marcan con la etiqueta especial -1.

Algoritmo

b) Hay una transición de un pixel del frente a un pixel sin marcar del fondo.

   Esto indica que el pixel está en un contorno interno.

   Comenzando en el pixel se recorre el contorno interno.

   A todos los pixeles del fondo que son vecinos se les asigna la etiqueta especial -1.

c) Si el pixel del frente no está en un contorno, el pixel a la izquierda está etiquetado y esa etiqueta se propaga al pixel actual.

Algoritm

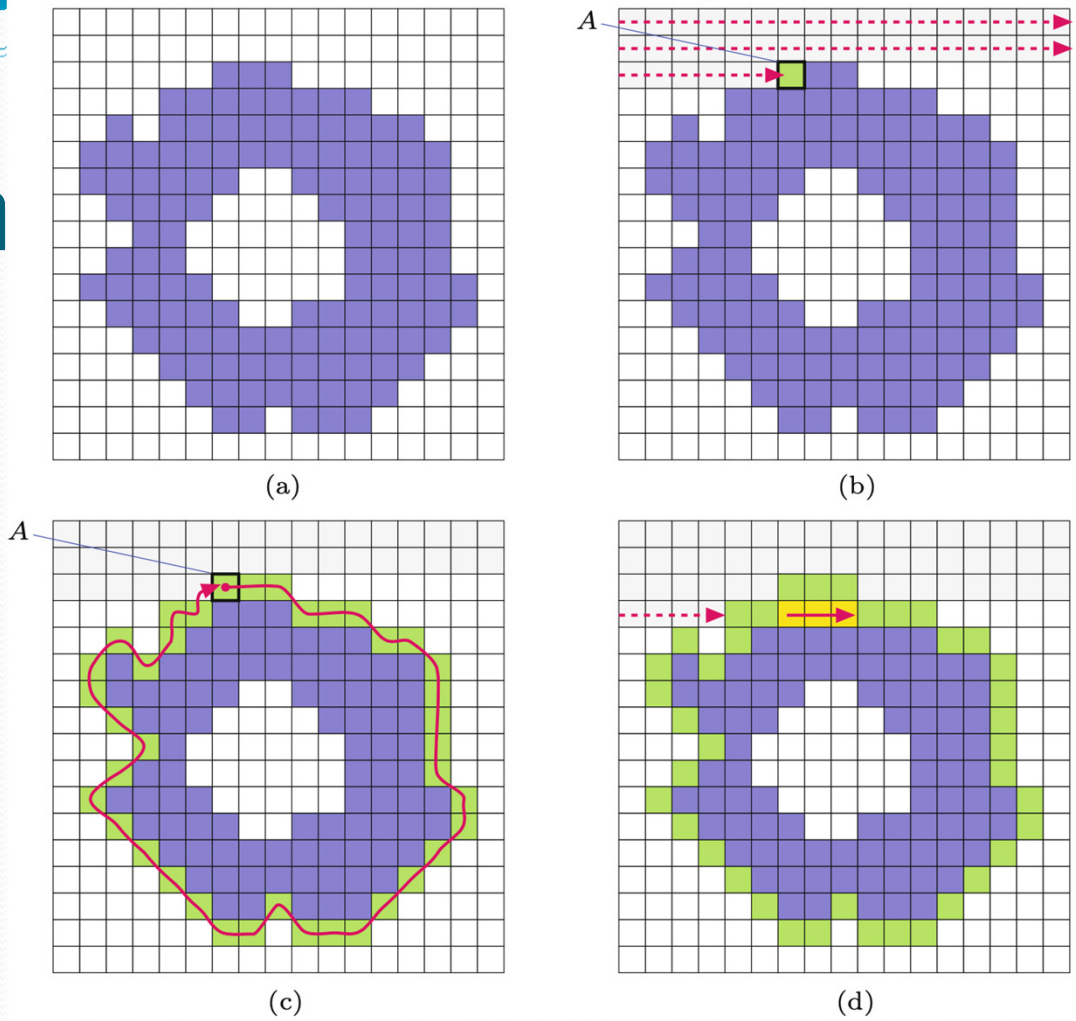
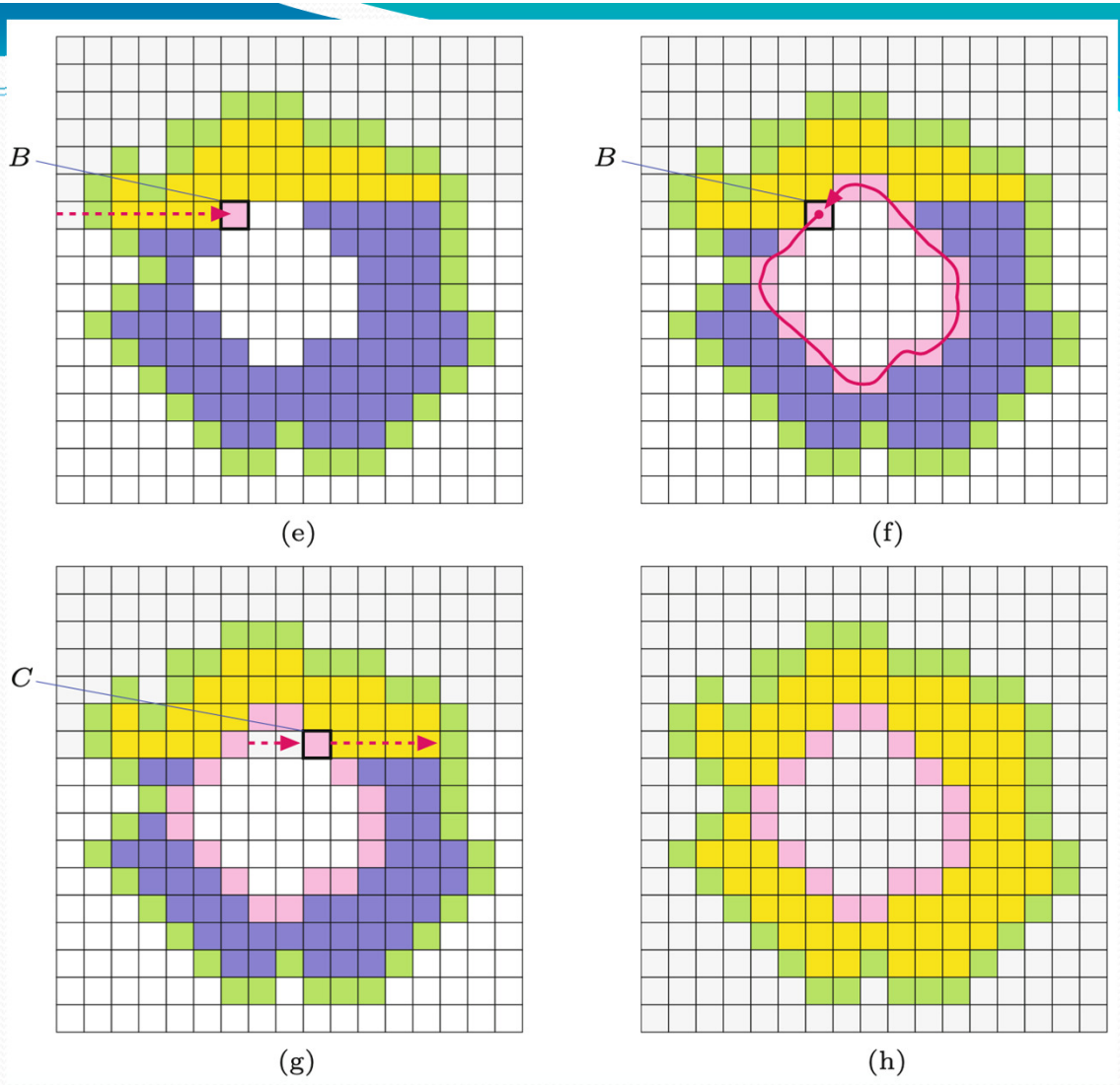


Fig. 10.9
Combined region labeling and contour following (after [47]). The image in (a) is traversed from the top left to the lower right, one row at a time. In (b), the first foreground pixel A on the outer edge of the region is found. Starting from point A, the pixels on the edge along the outer contour are visited and labeled until A is reached again (c). Labels picked up at the outer contour are propagated along the image line inside the region (d). In (e), B was found as the first point on the *inner contour*. Now the inner contour is traversed in clock-wise direction, marking the contour pixels until point B is reached again (f). The same tracing process is used as in step (c), with the inside of the region always lying to the right of the contour path. In (g) a previously marked point C on an inner contour is detected. Its label is again propagated along the image line inside the region. The final result is shown in (h).

Algoritmo



Algoritmo

- Conclusiones:
- El algoritmo es eficiente y no requiere grandes cantidades de memoria.
- Es adecuado para procesar imágenes binarias grandes.

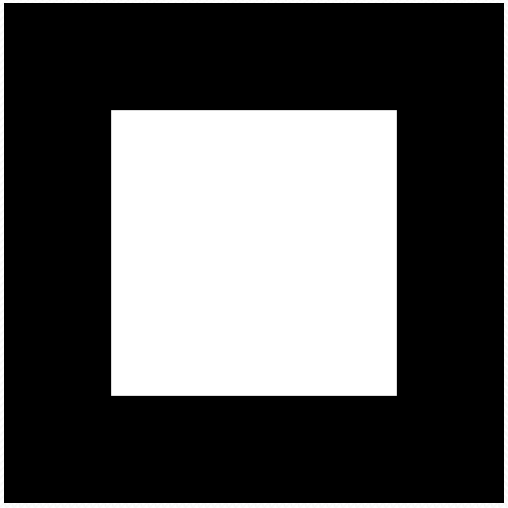
Detección de contornos en OpenCV

1. Leer una imagen.
2. Si la imagen es a color se convierte a tonos de gris con cv.cvtColor().
3. Opcionalmente, aplicar un filtro de blur.
4. Binarizar la imagen con cv.threshold().
5. Detectar los contornos con cv.findContours().
6. Dibujar los contornos con cv.drawContours().

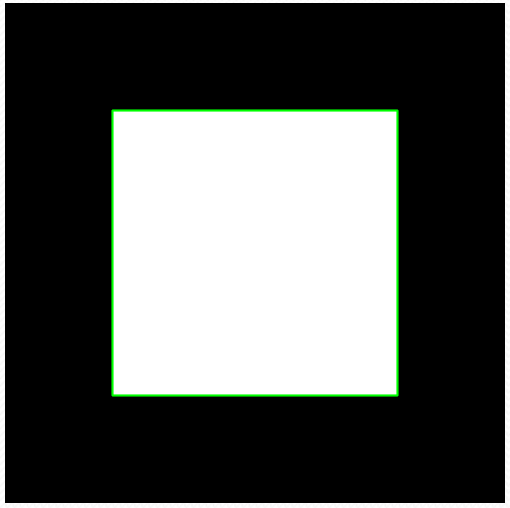
Ejemplo

- Archivo: object_detection/contour_simple.py.

Imagen original

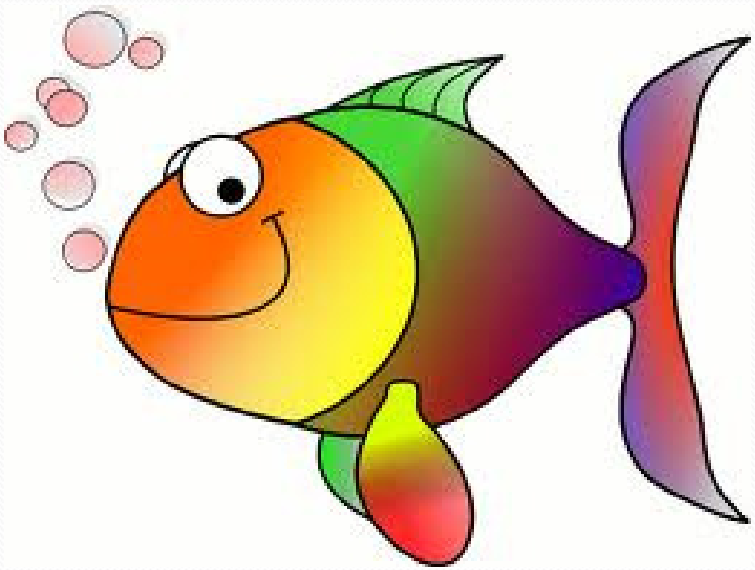


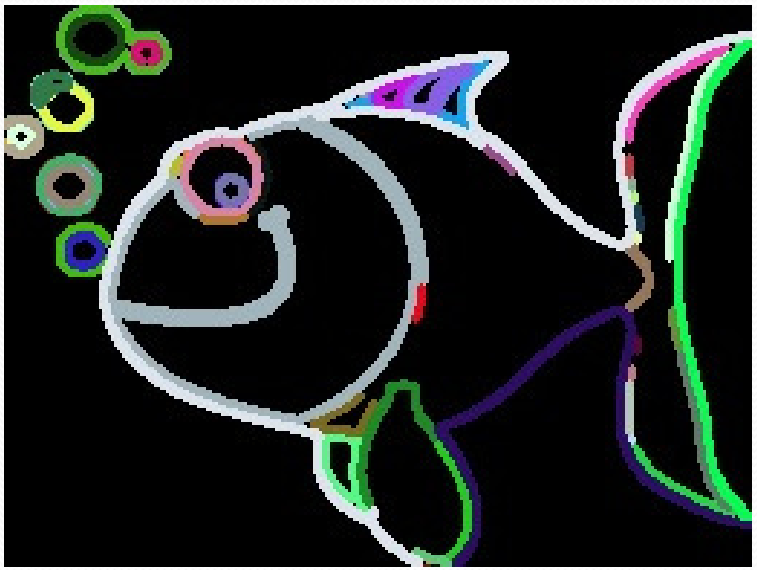
Contorno



Ejemplo

- Archivo: object_detection/contour_fish.py.





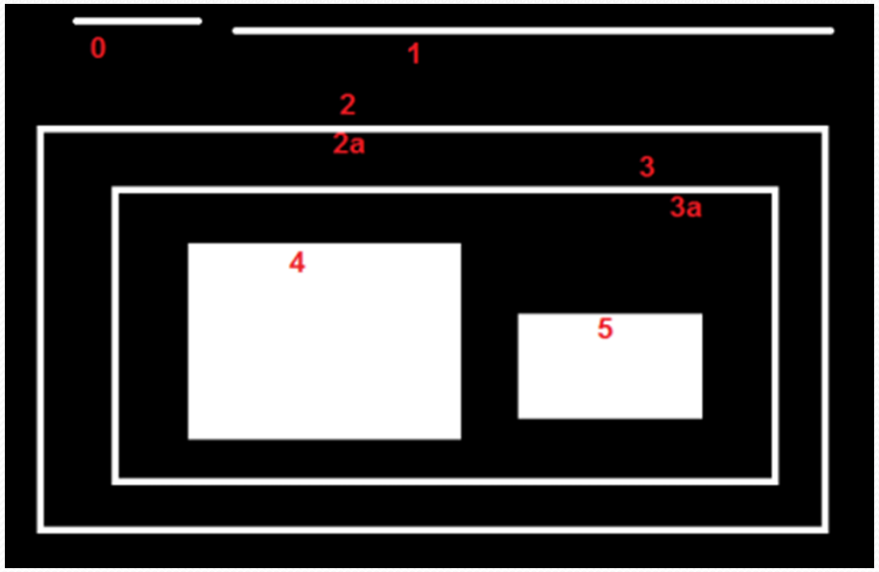
Función cv2.findContours()

- La función findContours() regresa dos arreglos:
- contours – contornos detectados
- hierarchy – información sobre la topología de la imagen

Jerarquía de contornos en OpenCV

- Cada contorno tiene información sobre su jerarquía, quién es su hijo, quién es su padre, etc.
- OpenCV lo representa en un arreglo de cuatro valores: [Next, Previous, First_child, Parent].
- Next – contorno siguiente al mismo nivel
- Previous – contorno anterior al mismo nivel
- First_child – primer contorno al siguiente nivel
- Parent – contorno padre

Modo de recuperación RETR_LIST



array([[[1, -1, -1, -1], [2, 0, -1, -1], [3, 1, -1, -1], [4, 2, -1, -1],
[5, 3, -1, -1], [6, 4, -1, -1], [7, 5, -1, -1], [-1, 6, -1, -1]]])

Modo de recuperación RETR_EXTERNAL

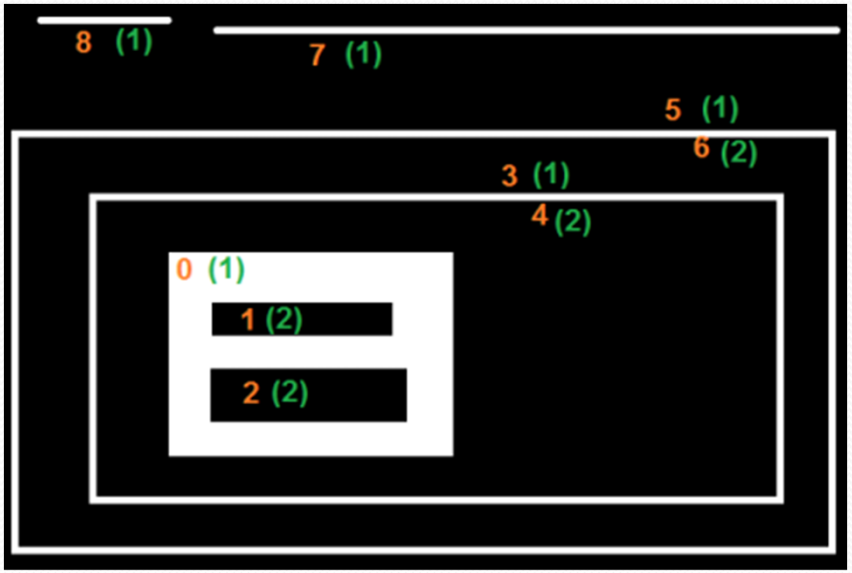
- Regresa solo los contornos externos.
- En la imagen anterior regresa los contornos 0, 1 y 2.

```
array([[[ 1, -1, -1, -1],
        [ 2,  0, -1, -1],
        [-1,  1, -1, -1]]])
```

Modo de recuperación RETR_CCOMP

- Recupera todos los contornos y los organiza en una jerarquía de 2 niveles.
- Los contornos externos del objeto se colocan en la jerarquía-1.
- Los contornos de los agujeros dentro del objeto (si los hay) se colocan en la jerarquía-2.
- Si hay algún objeto dentro de él, su contorno se coloca nuevamente en la jerarquía-1.
- Y su agujero en la jerarquía-2.
- Así sucesivamente.

Modo de recuperación RETR_CCOMP



array([[[3, -1, 1, -1], [2, -1, -1, 0], [-1, 1, -1, 0], [5, 0, 4, -1],
[-1, -1, -1, 3], [7, 3, 6, -1], [-1, -1, -1, 5], [8, 5, -1, -1], [-1, 7, -1, -1]]])

Modo de recuperación RETR_TREE

- Recupera todos los contornos y crea una lista con la jerarquía completa.

Modo de recuperación RETR_TREE

array([[[7, -1, 1, -1], [-1, -1, 2, 0], [-1, -1, 3, 1], [-1, -1, 4, 2],
[-1, -1, 5, 3], [6, -1, -1, 4], [-1, 5, -1, 4], [8, 0, -1, -1], [-1, 7, -1, -1]]])

Representación de imágenes binarias

- Representación de matriz.
- Codificación de longitud de corridas (RLE – run-length encoding).

Representación de matriz

- Cada elemento representa el color o intensidad en la posición dada por el renglón y la columna.
- En particular, en una imagen binaria cada elemento vale 0 o 1 (un bit).
- También se les llama bitmaps.
- La desventaja es que el espacio en memoria depende del tamaño de la imagen.
- No hace diferencia entre una imagen que contenga unos cuantos pixeles en 1 y una imagen con una escena compleja.

RLE

- Una corrida (run) es una secuencia contigua de pixeles del frente dentro del mismo renglón o columna.
- Cada corrida se codifica mediante 3 enteros:
- $Run_i = \langle renglón_i, columna_i, tamaño_i \rangle$

RLE

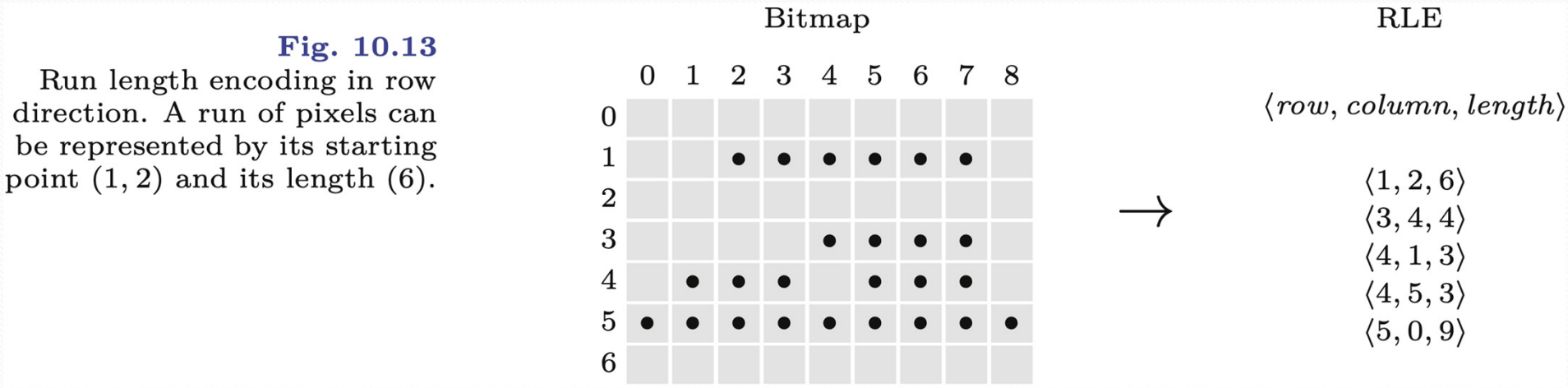


Fig. 10.13
Run length encoding in row direction. A run of pixels can be represented by its starting point $(1, 2)$ and its length (6).

RLE

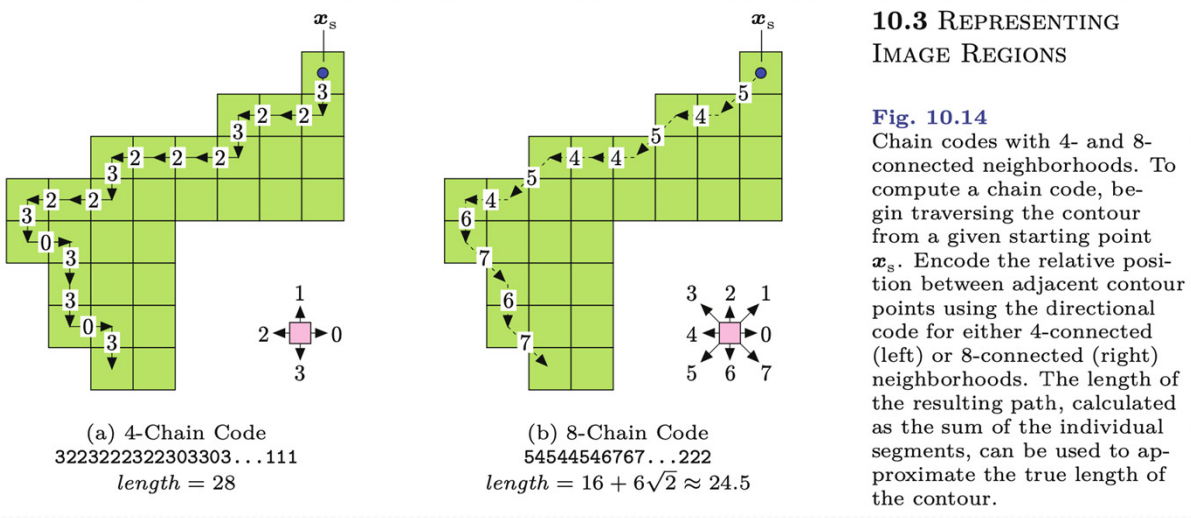
- RLE es fácil de implementar y eficiente de calcular.
- Se puede utilizar como método de compresión sin pérdidas (lossless).
- Se encuentra en algunos codecs como TIFF, JPEG y GIF.

Propiedades de las imágenes binarias

- Características geométricas:
  - Perímetro
  - Área
  - Compacidad
  - Redondez
  - Bounding box
  - Envolvente convexa (convex hull)
  - Convexidad
  - Densidad
  - Diámetro

Perímetro

- El perímetro (o circunferencia) de una región R se define como la longitud de su contorno exterior, donde R debe estar conectada.
- El tipo de vecindad debe tenerse en cuenta para este cálculo.



10.3 REPRESENTING IMAGE REGIONS

Fig. 10.14
Chain codes with 4- and 8-connected neighborhoods. To compute a chain code, begin traversing the contour from a given starting point x_s. Encode the relative position between adjacent contour points using the directional code for either 4-connected (left) or 8-connected (right) neighborhoods. The length of the resulting path, calculated as the sum of the individual segments, can be used to approximate the true length of the contour.

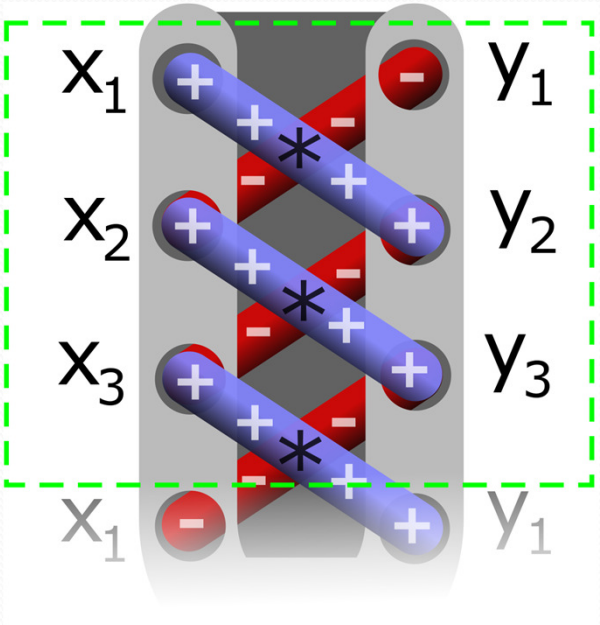
Perímetro

- Cuando se utilizan 4-vecinos, los segmentos se ponderan con 1.
- En el caso de 8-vecinos, una buena aproximación es ponderar los segmentos horizontal y vertical con 1 y los segmentos diagonales con $\sqrt{2}$.
- En el cálculo del perímetro de una imagen binaria se recomienda usar el factor de corrección 0.95.

Área

- El área de una región binaria R se encuentra contando el número de pixeles en dicha región.
- $A(R) = N = |R|$
- Para una región conectada sin hoyos, con un contorno definido por M puntos $(u_0, v_0), (u_1, v_1), \ldots, (u_{M-1}, v_{M-1})$, el área se puede aproximar mediante la fórmula de Gauss para polígonos:
- $A(R) \approx \frac{1}{2} \cdot \left[\sum_{i=0}^{M-1}\left(u_i \cdot v_{(i+1) \bmod M} - u_{(i+1) \bmod M} \cdot v_i\right)\right]$

Fórmula de Gauss



By Job Bouwman - Own work, CC BY-SA 4.0,
https://commons.wikimedia.org/w/index.php?curid=55566629

Área

- El perímetro y el área son son invariantes a las traslaciones o las rotaciones.
- Si se afectan por escalamientos.

Compacidad

- En inglés *compactness*.
- La compacidad de una región R se define como
- $C(R) = \frac{A(R)}{P(R)}$
- El área crece cuadráticamente mientras que el perímetro lo hace linealmente.
- Para un círculo: $A(R) = \pi \cdot r^2$, $P(R) = 2\pi \cdot r$.
- Para un cuadrado: $A(R) = L^2$, $P(R) = 4 \cdot L$.

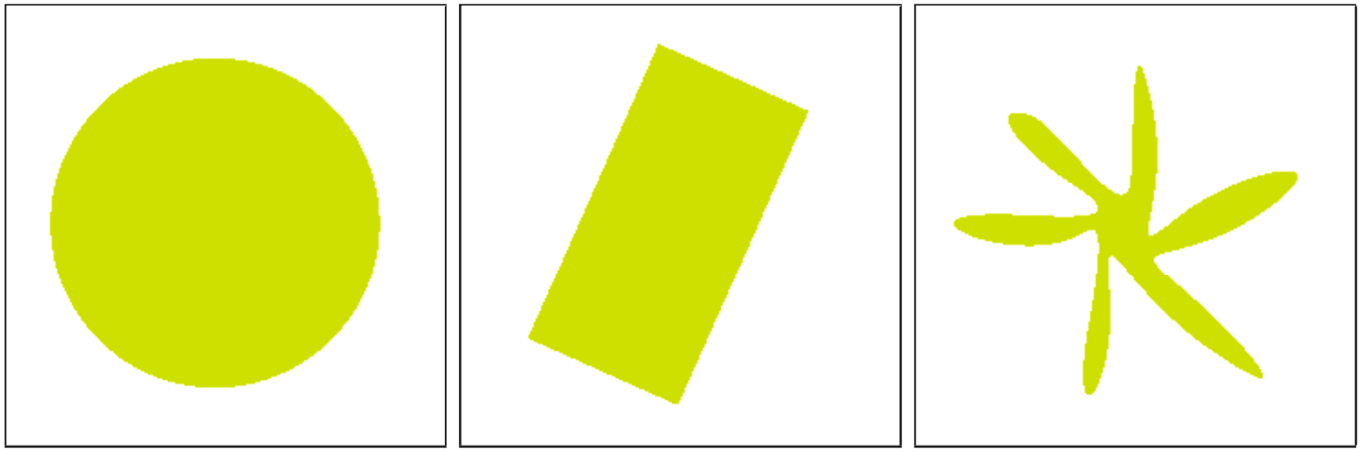
Compacidad

- La compacidad es invariante a la traslación, rotación y escalamiento.
- Para un círculo, esta relación es $\frac{A(R)}{P^2(R)} = \frac{\pi \cdot r^2}{4\pi \cdot r^2} = \frac{1}{4\pi}$.
- Para un cuadrado es $\frac{A(R)}{P^2(R)} = \frac{L^2}{16 \cdot L^2} = \frac{1}{16}$.

Redondez

- Mide que tan redonda (o circular) es una región.
- $\text{Circularity}(R) = 4\pi \cdot \frac{A(R)}{P^2(R)}$
- Para un círculo vale 1 y para otras figuras está en el rango [0, 1].
- Para un cuadrado: $\text{Circularity}(R) = 4\pi \cdot \frac{1}{16} = \frac{\pi}{4} \approx 0.79$.
- Recordar que en el cálculo del perímetro de una imagen binaria se recomienda usar el factor de corrección 0.95.

Redondez



(a) 0.904
(1.001)

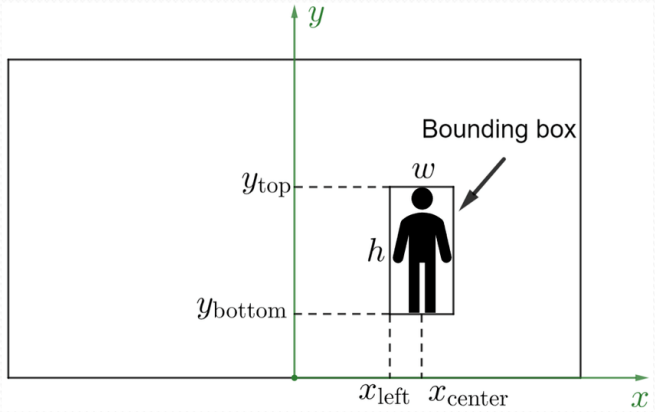
(b) 0.607
(0.672)

(c) 0.078
(0.086)

Fig. 10.15
Circularity values for different shapes. Shown are the corresponding estimates for Circularity($\mathcal{R}$) as defined in Eqn. (10.15). Corrected values calculated with Eqn. (10.12) are shown in parentheses.

Bounding box

- El bounding box de una región R es el rectángulo mínimo, paralelo a los ejes, que encierra todos los puntos de R.

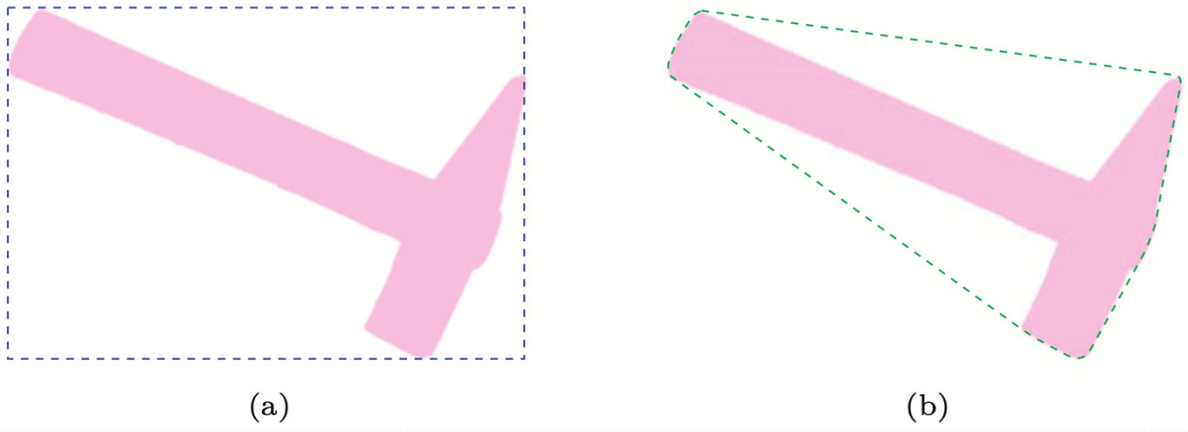


Fuente:license: Creative Commons Attribution 4.0 International
https://www.researchgate.net/publication/356632069/figure/fig3/AS:1095686288347139@1638243376533/Bounding-box-definition-in-the-image-coordinates-system.ppm

Envolvente convexa

- En inglés *convex hull*.
- La envolvente convexa es el polígono convexo más pequeño que contiene todos los puntos de la región R.

Fig. 10.16
Example bounding box (a) and convex hull (b) of a binary image region.



Convexidad

- Es la relación entre la longitud de la envolvente convexa y el perímetro de la región.
- $\text{Convexity}(R) = \frac{|H(R)|}{P(R)}$
- Para un círculo: $\text{Convexity}(R) > 1$.
- Para un cuadrado: $\text{Convexity}(R) = 1$.

Densidad

- La densidad es la relación entre el área de la región y el área de su envolvente conexa.
- $\text{Density}(R) = \frac{A(R)}{A_H(R)}$
- Para un círculo: $\text{Density}(R) < 1$.
- Para un cuadrado: $\text{Density}(R) = 1$.

Diámetro

- El diámetro, es la distancia máxima entre dos puntos cualesquiera en la envolvente convexa.