08.10.2021

Digital Image Processing (CSE/ECE 478)
Lecture-13: Morphological Operations, Intro to Geometric Operations

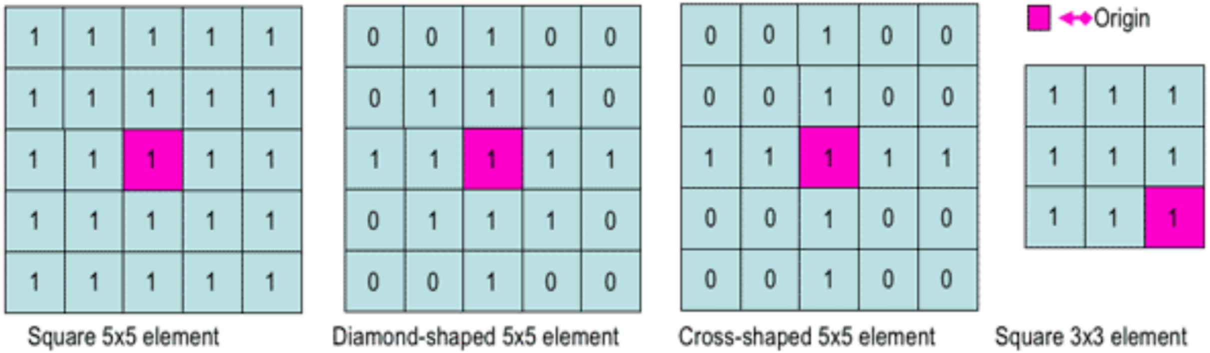
Ravi Kiran and Sudipta Banerjee

CVIT

Structuring Element

The **structuring element** is a small binary image, i.e. a small matrix of pixels, each with a value of zero or one:

- The matrix dimensions specify the *size* of the structuring element.
- The pattern of ones and zeros specifies the *shape* of the structuring element.
- An *origin* of the structuring element is usually one of its pixels, although generally the origin can be outside the structuring element.



Examples of simple structuring elements.

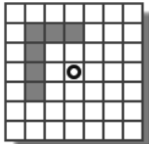


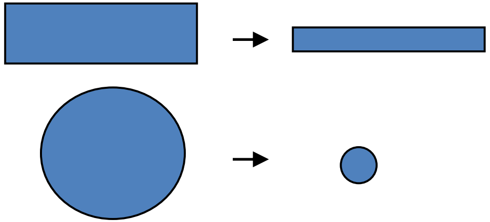
Image Courtesy: https://www.cs.auckland.ac.nz/courses/compsci773s1c/lectures/ImageProcessing-html/topic4.htm

Erosion

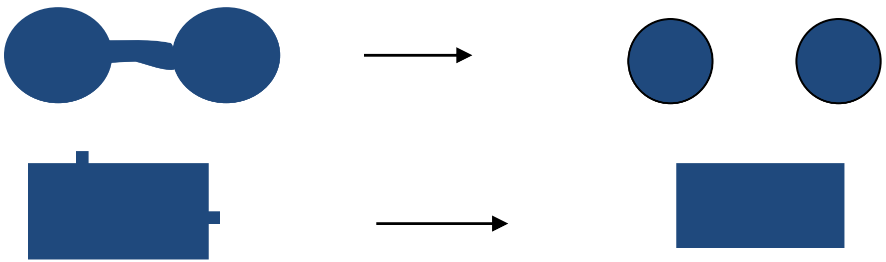
Erosion shrinks the connected sets of 1s of a binary image.

It can be used for

1. shrinking features



2. Removing bridges, branches and small protrusions



Erosion

- Shrinks foreground objects
- Foreground holes are enlarged
- Representation: $f \ominus s$ (f: binary image, s: SE)

Erosion : Operation (min filter)

1	1	1
1	1X	1
1	1	1

Set of coordinate points =

{ (-1, -1), (0, -1), (1, -1),

(-1, 0), (0, 0), (1, 0),

(-1, 1), (0, 1), (1, 1) }

If, for a particular location of Structuring Element (SE) origin, SE lies **fully within the region**, retain the location, else set to 0

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0
0	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0
0	0	1	1	1	1	1	0	0	0	0	1	1	1	0	0
0	0	1	1	1	1	0	0	0	0	1	1	1	1	0	0
0	0	0	1	1	0	0	0	0	1	1	1	1	1	0	0
0	0	0	0	0	0	0	0	1	1	1	1	1	0	0	0
0	0	0	0	0	0	0	1	1	1	1	1	0	0	0	0
0	0	0	0	0	0	1	1	1	1	1	0	0	0	0	0
0	0	0	0	0	1	1	1	1	1	0	0	0	0	0	0
0	0	0	0	1	1	1	1	1	0	0	0	0	0	0	0
0	0	0	0	1	1	1	1	1	1	1	1	0	0	0	0
0	0	0	0	1	1	1	1	1	1	1	1	0	0	0	0
0	0	0	0	0	1	1	1	1	1	1	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

$\Rightarrow$

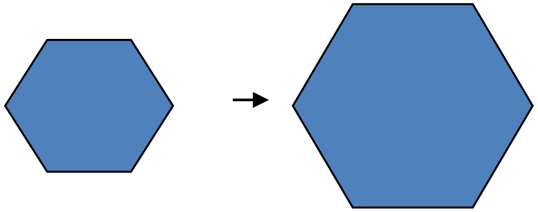
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0
0	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0
0	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Dilation

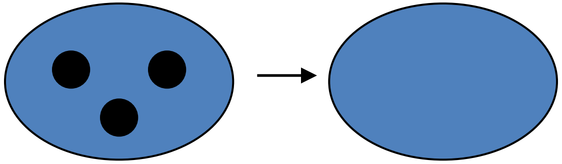
Dilation expands the connected sets of 1s of a binary image.

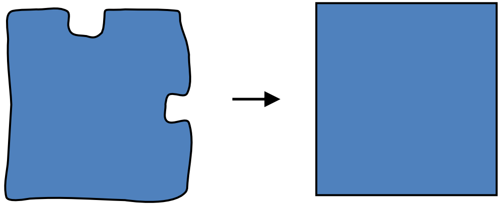
It can be used for

1. growing features



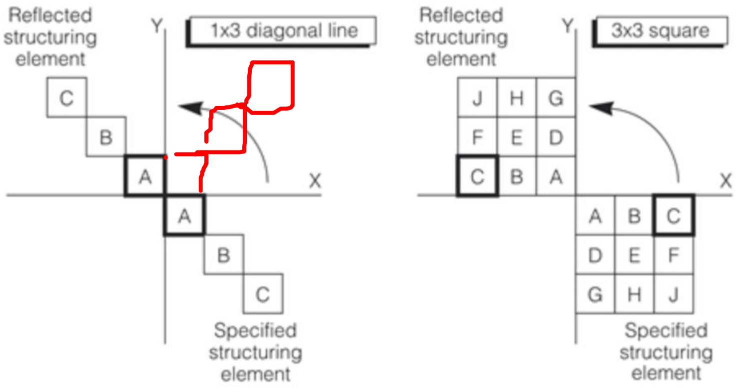
2. filling holes and gaps





Dilation

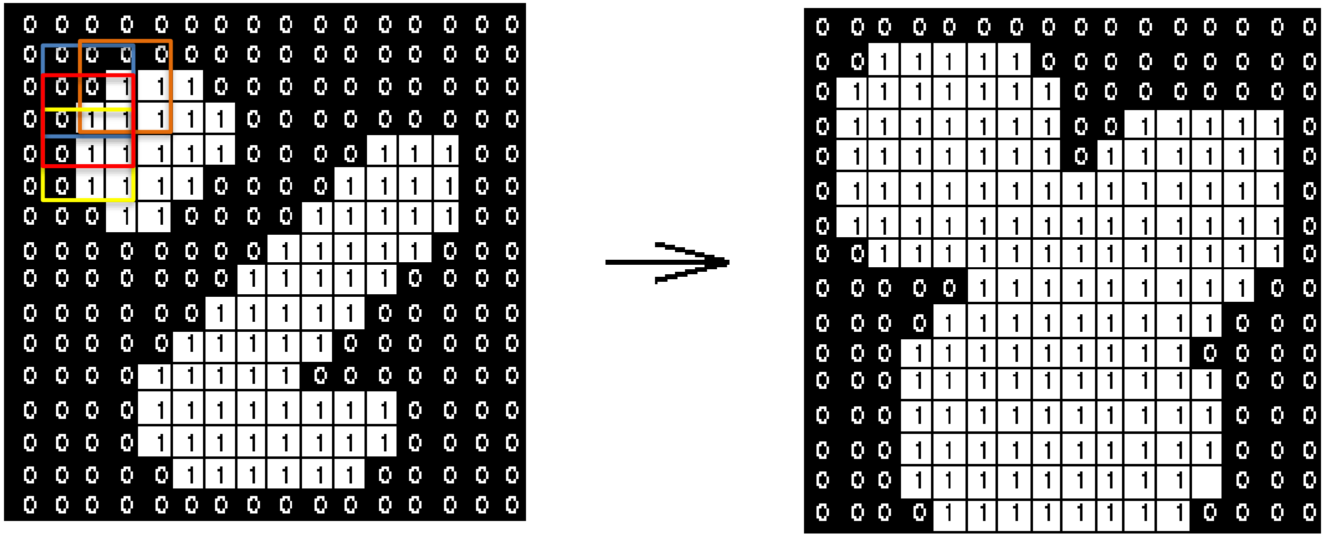
- Expands foreground objects
- Foreground holes are shrunk
- Representation: $f \oplus \hat{s}$ (f: binary image, $\hat{s}$: Reflected version of SE about its origin)



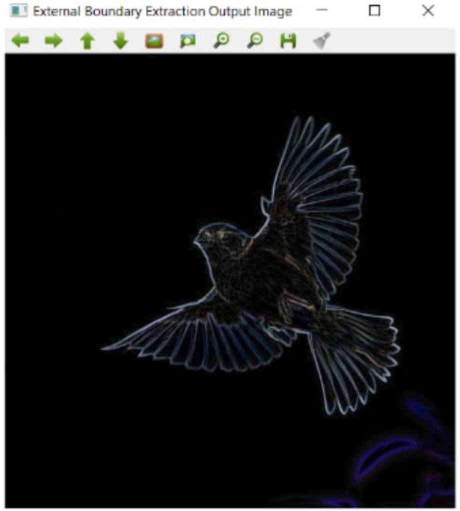
Dilation (max filter)

1	1	1
1	1 X	1
1	1	1

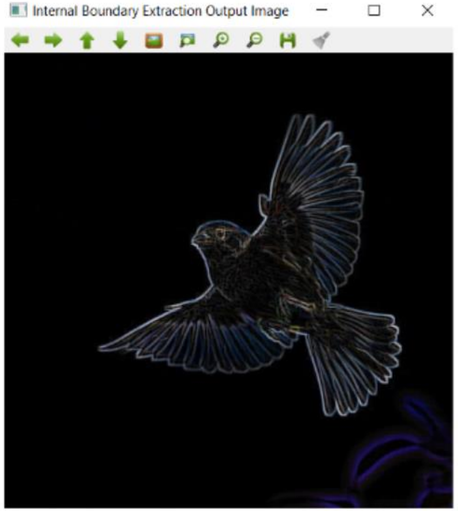
1. First reflect the SE about its origin
2. Move the SE within foreground
3. Check if the SE intersects with foreground of the binary image:
4. If the origin intersects, add the remaining background pixels as foreground



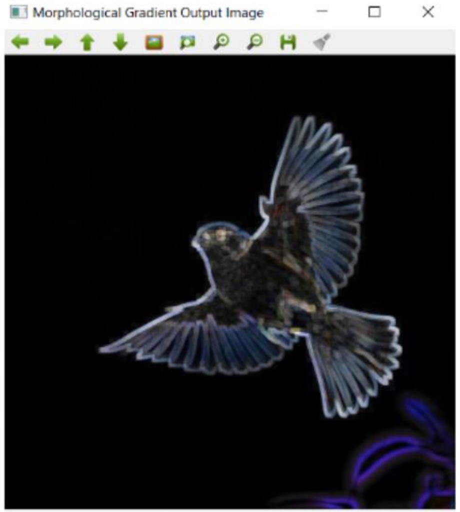
Boundary extraction



$(A \oplus B)-A$



$A-(A \ominus B)$



$(A \oplus B)-(A \ominus B)$

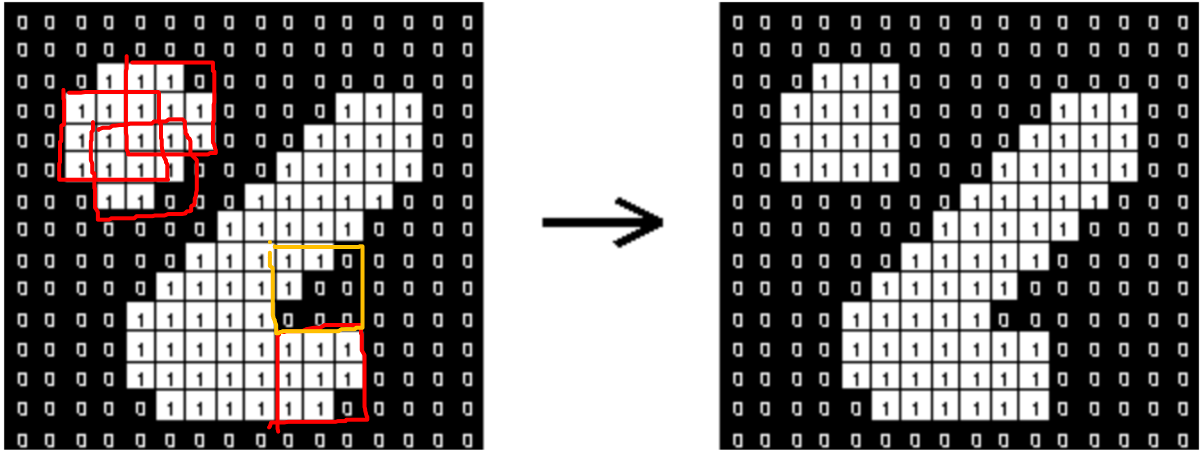
A: Original image; B: SE square of size 7x7

https://towardsdatascience.com/image-processing-part-3-dbf103622909

Opening (Erosion then Dilation)

- Take the structuring element (SE) and <u>slide it around *inside* each foreground region</u>.
  - All foreground pixels which can be covered by the SE with the SE being entirely within the foreground region will be preserved.
  - All foreground pixels which can *not* be reached by the structuring element without lapping over the edge of the foreground object will be eroded away!

Removes small objects /noise from FG



SE: 3x3 square

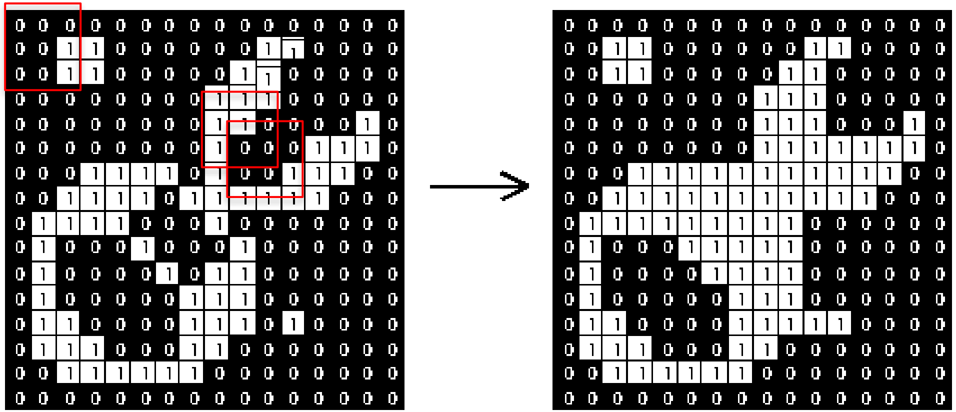
https://homepages.inf.ed.ac.uk/rbf/HIPR2/open.htm

Closing (Dilation then Erosion)

- Take the structuring element (SE) and <u>slide it around *outside* each foreground region</u>.
  - For any background pixel, if the SE *can touch* it without any part of the SE being inside the foreground region, the pixel stays as background
  - Any background pixel that *cannot* be touched by SE without coming inside the foreground region is changed to become a foreground pixel

Removes or closes small holes in FG

SE: 3x3 square



Morphological Smoothing

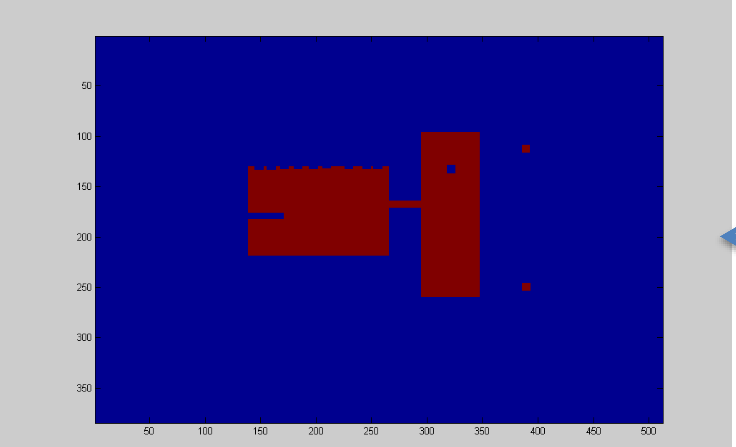
Opening followed by closing

(a) (b)

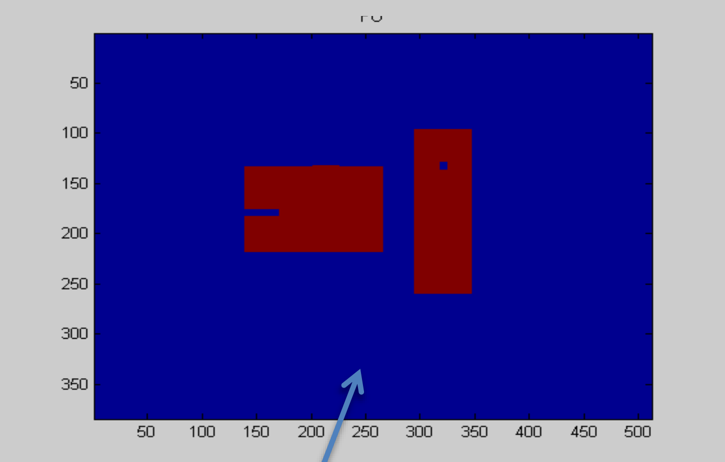
(a) Original gray image. (b) Morphological smoothed image.

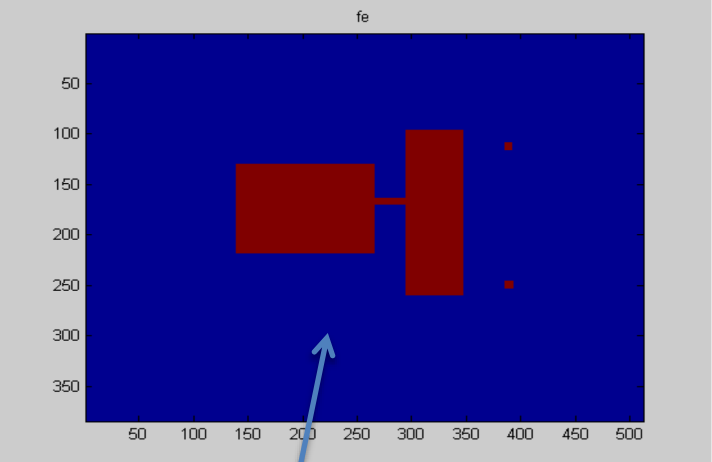
Image courtesy: https://www.uotechnology.edu.iq/ce/Lectures/Image_Processing_4th/DIP_Lecture11.pdf

Read: http://support.ptc.com/help/mathcad/en/index.html#page/PTC_Mathcad_Help/example_grayscale_morphology.html



ORIGINAL





OPENING $f \circ s = (f \ominus s) \oplus s$ **CLOSING** $f \cdot s = (f \oplus s) \ominus s$

Idempotent; Dual

MAGNITUDE RELATIONS

- Dilation and closing are *extending operations*, meaning that foreground pixels are added to the image.
- Erosion and opening are *narrowing operations*, meaning that foreground pixels are removed.
- For a binary image f and a binary structuring element s, we have that

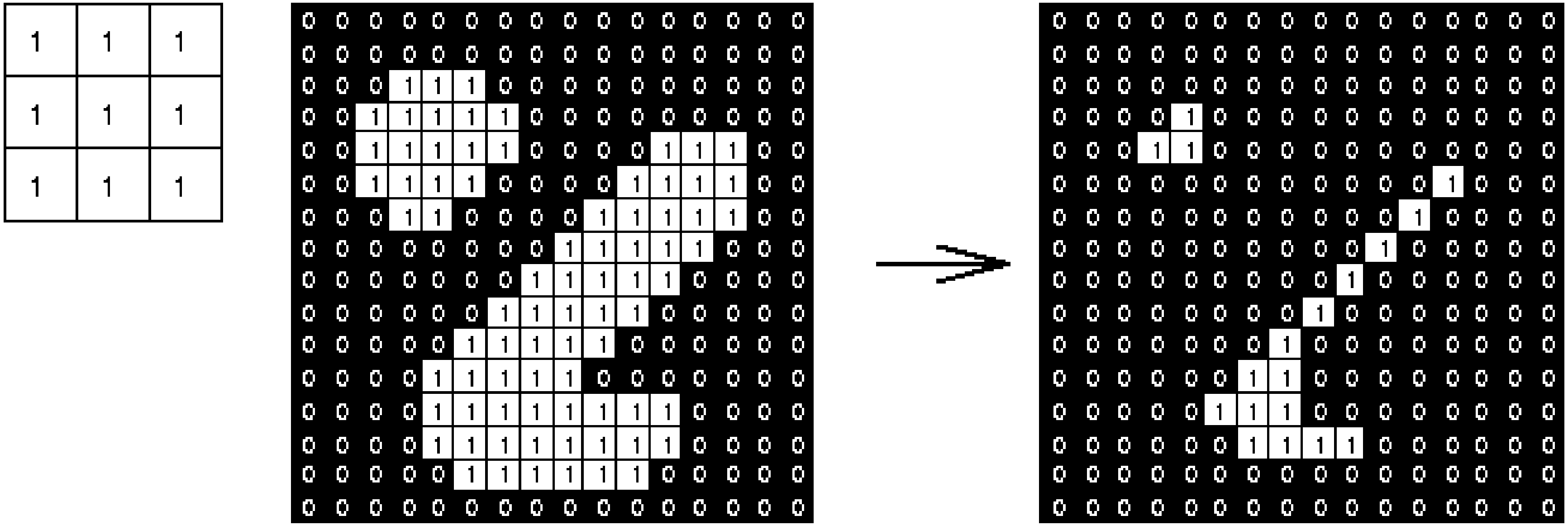
$$(f \ominus s)(x) \leq (f \circ s)(x) \leq f(x) \leq (f \bullet s)(x) \leq (f \oplus s)(x)$$

- On a similar note, if $F(g)$ is the set of foreground pixels in g,

$$F(f \ominus s) \subseteq F(f \circ s) \subseteq F(f) \subseteq F(f \bullet s) \subseteq F(f \oplus s)$$

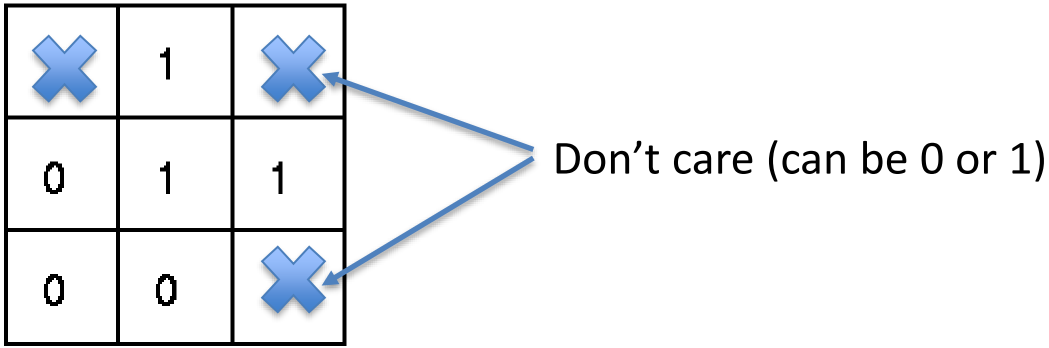
Erosion

- Simple application of **pattern matching**
  - **Fixed template**



Hit-or-miss Transform

- Look for particular patterns of foreground and background pixels.



- If matched, set pixel = 1

Example: Find right-angled convex corners

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	1	1	1	1	1	0	0	0
0	0	1	1	1	1	0	0	1	1	1	1	1	1	0	0
0	0	1	1	1	1	0	0	1	1	1	1	1	0	0	0
0	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0
0	0	1	1	1	1	1	1	1	1	1	1	0	0	0	0
0	0	1	1	0	0	0	0	0	1	1	1	1	0	0	0
0	0	1	1	0	0	0	0	0	1	1	1	1	0	0	0
0	0	0	1	0	0	0	0	0	1	1	1	1	0	0	0
0	0	1	1	1	1	1	1	1	1	1	1	0	0	0	0
0	0	1	1	1	1	1	1	1	1	1	0	0	0	0	0
0	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0
0	0	1	1	1	1	1	1	1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

→

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0
0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
0	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0
0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
0	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Example: Find right-angled convex corners

	1	
0	1	1
0	0	

	1	
1	1	0
	0	0

	0	0
1	1	0
	1	

0	0	
0	1	1
	1	

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	1	1	1	1	1	0	0	0
0	0	1	1	1	1	0	0	1	1	1	1	1	1	0	0
0	0	1	1	1	1	0	0	1	1	1	1	1	0	0	0
0	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0
0	0	1	1	1	1	1	1	1	1	1	1	0	0	0	0
0	0	1	1	0	0	0	0	0	1	1	1	1	0	0	0
0	0	1	1	0	0	0	0	0	1	1	1	1	0	0	0
0	0	0	1	0	0	0	0	0	1	1	1	1	0	0	0
0	0	1	1	1	1	1	1	1	1	1	1	0	0	0	0
0	0	1	1	1	1	1	1	1	1	1	0	0	0	0	0
0	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0
0	0	1	1	1	1	1	1	1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

$\longrightarrow$

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0
0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
0	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0
0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
0	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Hit or Miss Transformation Alternative

The hit-or-miss transformation of an image A by B is denoted by $A \circledast B$.

B is a pair of structuring elements $B = (B_1, B_2)$ rather than a single element.

$B1$: set of elements of B associated with an object

$B2$: set of elements of B associated with the background

The hit-or-miss transform is defined as follows:

$$A \circledast B = (A \ominus B_1) \cap (A^c \ominus B_2)$$

This transform is useful in locating all pixel configurations that match the B_1 structure (i.e a hit) but do not match that of B_2 (i.e. a miss). Thus, the hit-or-miss transform is used for shape detection.

https://www.uotechnology.edu.iq/ce/Lectures/Image_Processing_4th/DIP_Lecture11.pdf

Sample Hit/Miss transforms

0	1	0
1	0	1
0	1	0

0	0	0
0	1	0
0	0	0

0	1	0
1	-1	1
0	1	0

Look for a pattern in which central pixel belongs to background, while top, left, right and bottom pixels belong to the foreground. ‘0’ is don’t care term in this example

Structuring elements (kernels). Left: kernel to 'hit'. Middle: kernel to 'miss'. Right: final combined kernel

0	0	0	0	0	0	0	0
0	255	255	255	0	0	0	255
0	255	255	255	0	0	0	0
0	255	255	255	0	255	0	0
0	0	255	0	0	0	0	0
0	0	255	0	0	255	255	0
0	255	0	255	0	0	255	0
0	255	255	0	0	0	0	0

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	255	0	0	0	0	0
0	0	0	0	0	0	0	0

0	-1	-1
1	1	-1
0	1	0

0	0	0	0	0	0	0	0
0	0	0	255	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	255	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

http://amroamroamro.github.io/mexopencv/opencv/hitmiss_demo.html

Sample Hit/Miss transforms

1)

0	0	0
0	1	0
0	0	0

Locate isolated points

2)

0	1	0
0	0	0

Locate end points on thin/tapering structures

3a)

	1	
	1	
1		1

3b)

1		
	1	
1		1

3c)

*	0	1
1	1	0
*	1	*

Locate triple junctions

Locate:

- Isolated foreground pixels (no neighboring foreground pixels)
- Foreground endpoints (one or zero neighboring foreground pixels)
- Foreground contours and junctions (atleast one neighboring foreground pixel)

Distance Transform (with chessboard distance metric)

Intensities in foregrounds now show the distance from each point to the closest background/boundary pixel

0	0	0	0	0	0	0	0
0	1	1	1	1	1	1	0
0	1	1	1	1	1	1	0
0	1	1	1	1	1	1	0
0	1	1	1	1	1	1	0
0	1	1	1	1	1	1	0
0	0	0	0	0	0	0	0

→

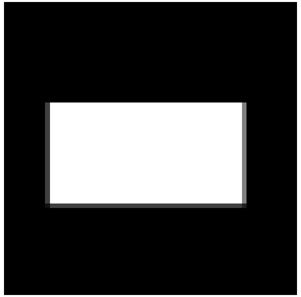
0	0	0	0	0	0	0	0
0	1	1	1	1	1	1	0
0	1	2	2	2	2	1	0
0	1	2	3	3	2	1	0
0	1	2	2	2	2	1	0
0	1	1	1	1	1	1	0
0	0	0	0	0	0	0	0

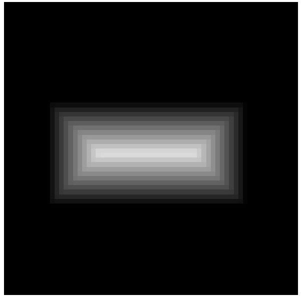
For each location x, find nearest point to P

Here, P is set of background pixels

$$DT(P)[x] = \min_{y \in P} D(x, y)$$

$$D_{chessboard}(\{a, b, c\}, \{x, y, z\}) = \max\{|a - x|, |b - y|, |z - c|\}$$





Distance Transform

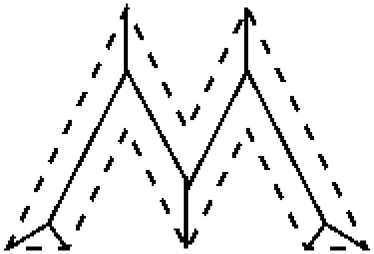
How to compute distance transform of binary image?

- Perform multiple successive erosions with suitable SE until all foreground regions in image are eroded
- Label each pixel with the number of erosions it took to disappear – Distance transform
- Simple but inefficient (Rosenfeld and Pfaltz 1968 recursive 2 pass morphology)

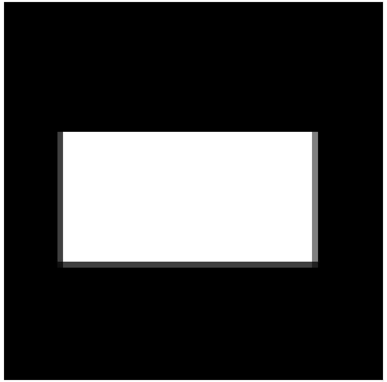
3x3 square SE: Chessboard; Cross-shaped SE: Cityblock; Disc-shaped SE: Euclidean

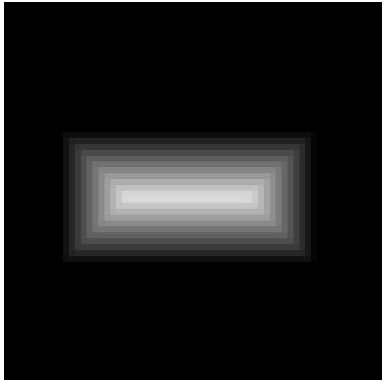
https://homepages.inf.ed.ac.uk/rbf/HIPR2/distance.htm

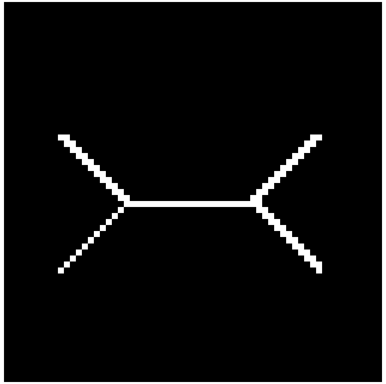
Skeletonization



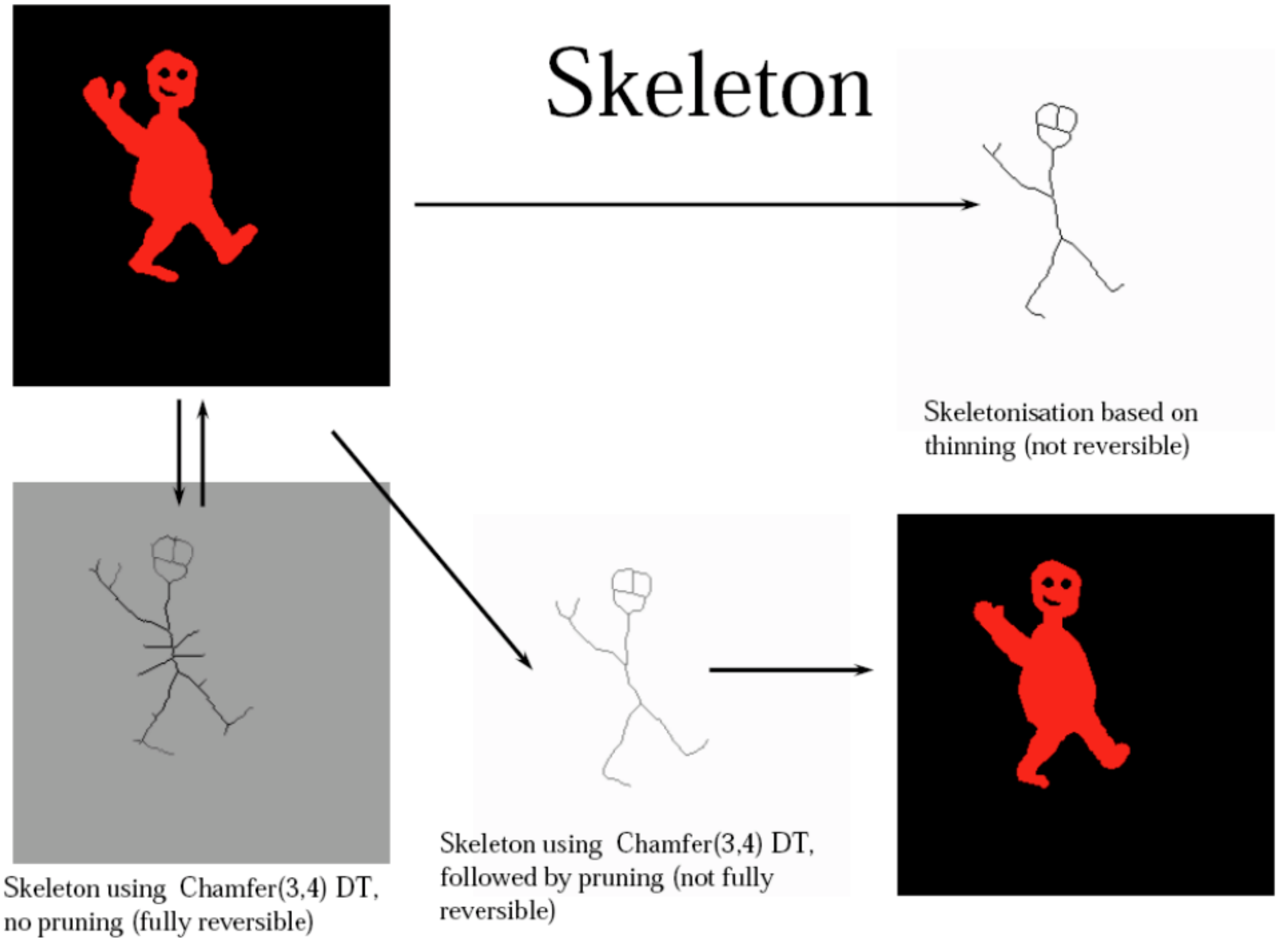
Process of reducing foreground regions to skeletal remnant (medial axis) preserving extent and connectivity of original region



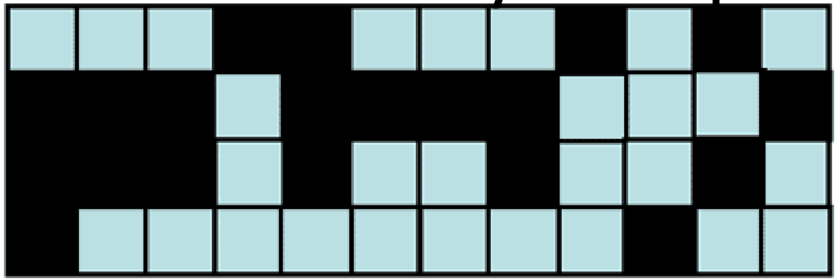




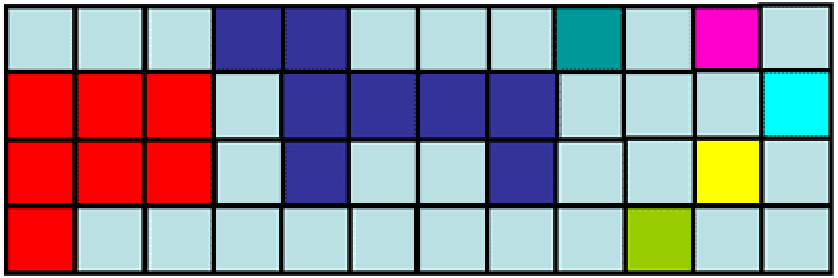
Skeleton


Skeleton
Skeletonisation based on thinning (not reversible)
Skeleton using Chamfer(3,4) DT, no pruning (fully reversible)
Skeleton using Chamfer(3,4) DT, followed by pruning (not fully reversible)

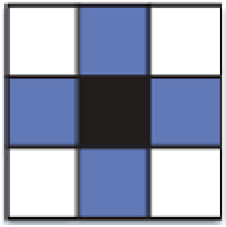
Can you tell how many ‘components’ in FG?

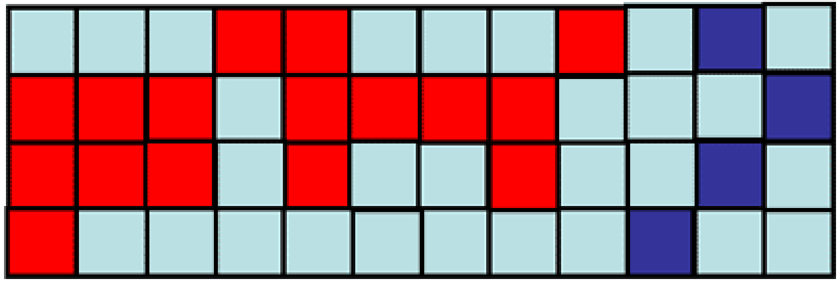


Binary image:
0 - objects;
1 -background

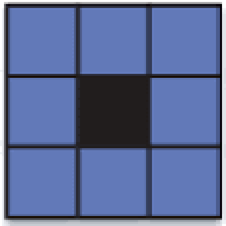


4-connected
objects +
8-connected
background



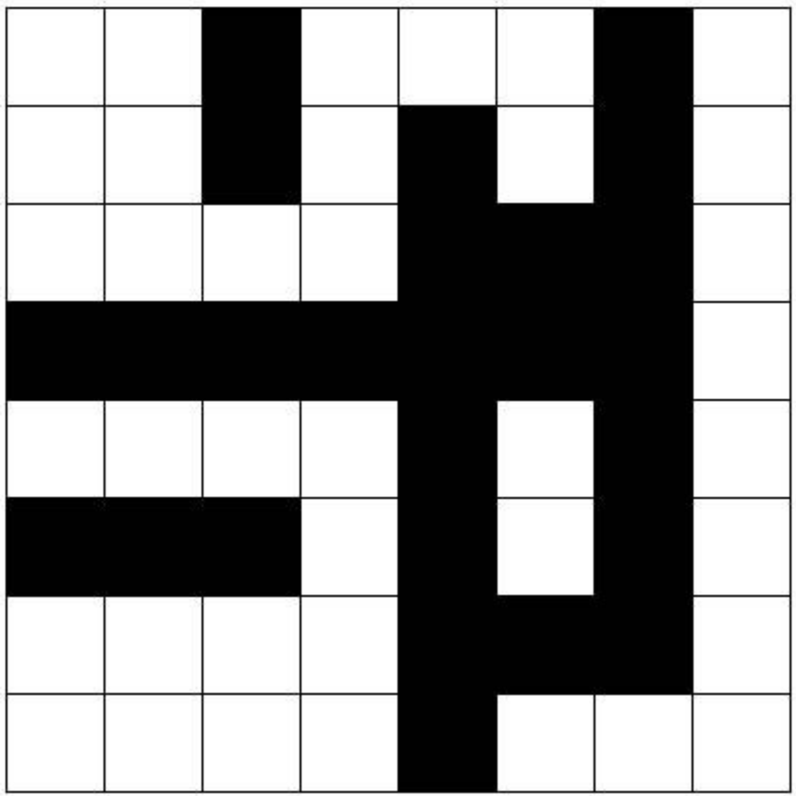


8-connected
objects +
8-connected
background

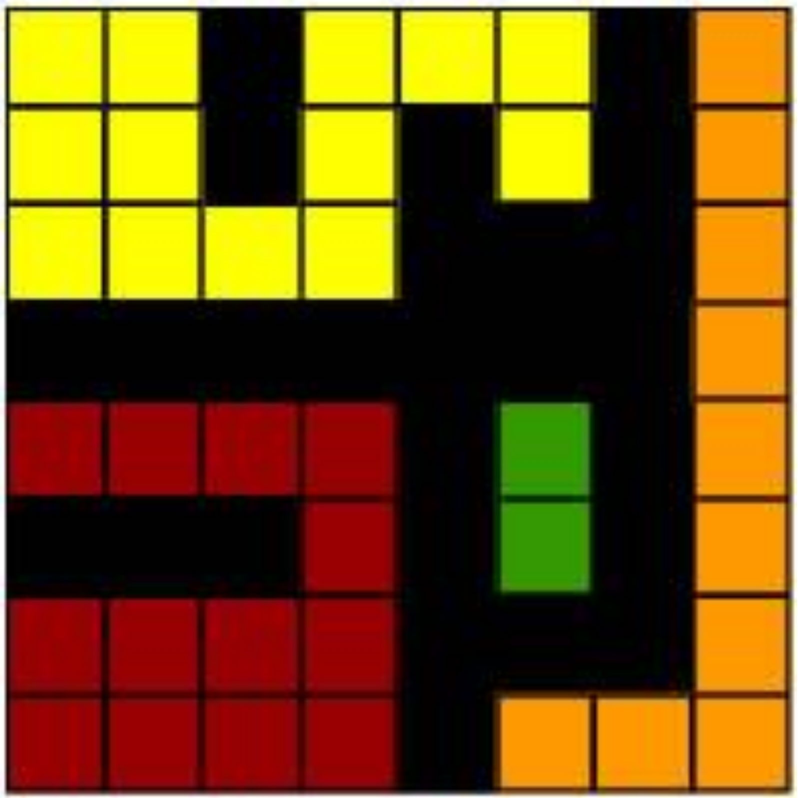


Two-Pass Algorithm for Connected Component Labelling

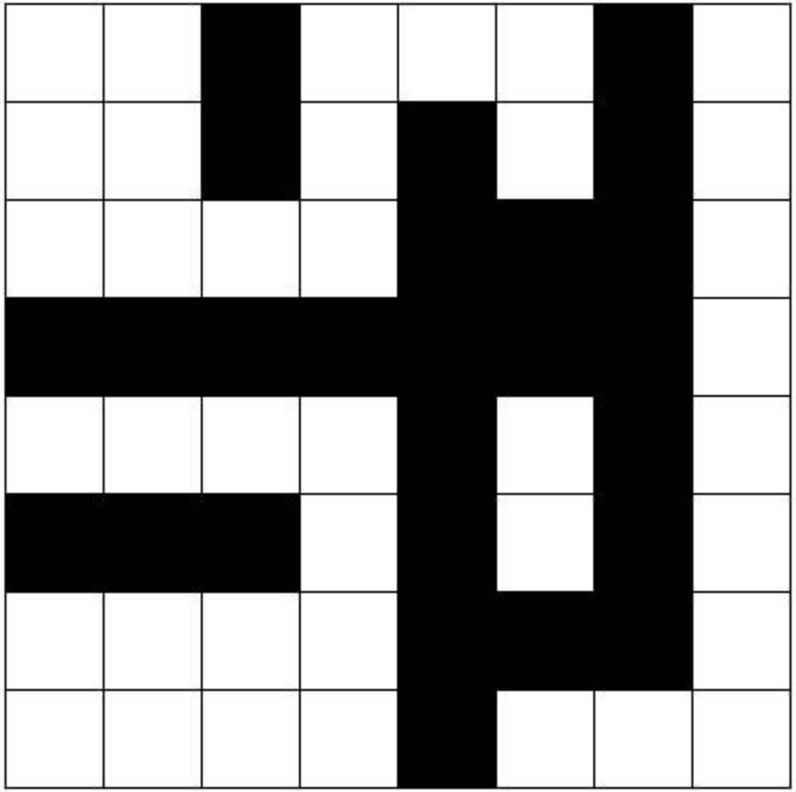
White is FG, black is BG



Two-Pass Algorithm for CCL

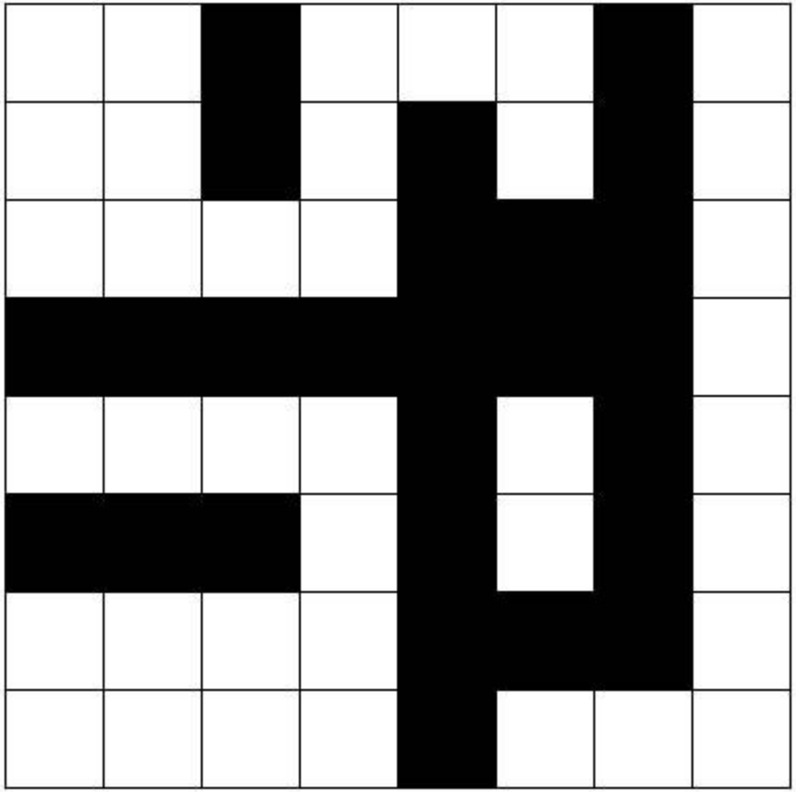


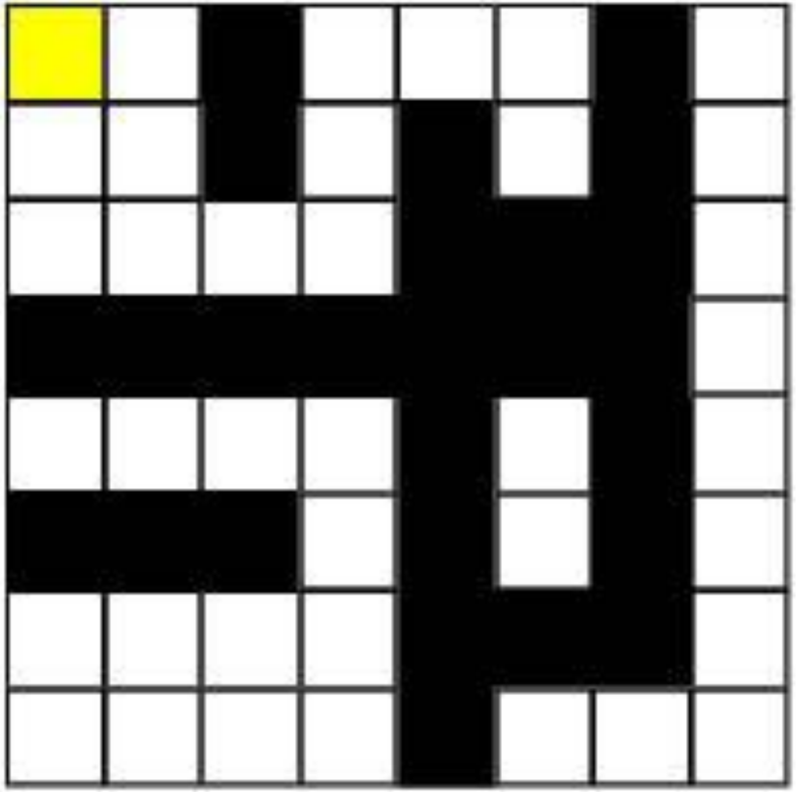
Two-Pass Algorithm for Connected Component Labelling



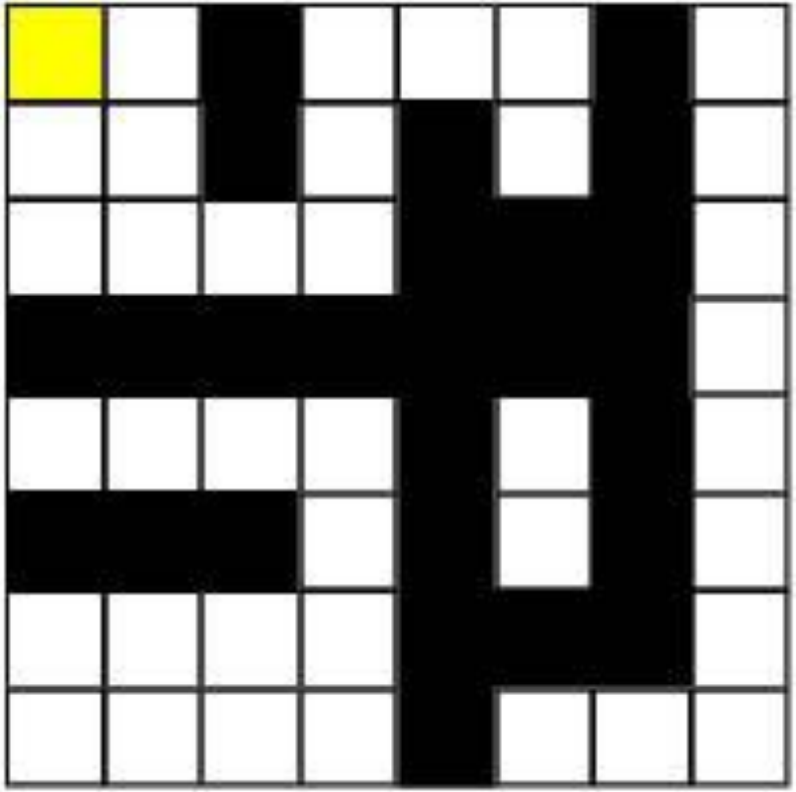
1	1	0	1	1	1	0	1
1	1	0	1	0	1	0	1
1	1	1	1	0	0	0	1
0	0	0	0	0	0	0	1
1	1	1	1	0	1	0	1
0	0	0	1	0	1	0	1
1	1	1	1	0	0	0	1
1	1	1	1	0	1	1	1

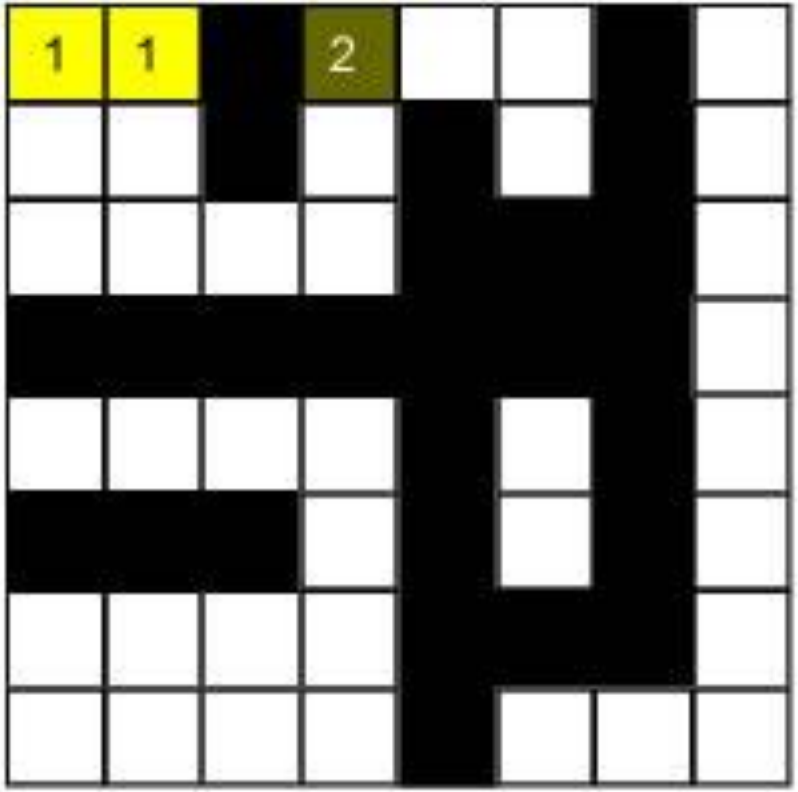
2PA: No top, left pixels → Create new label



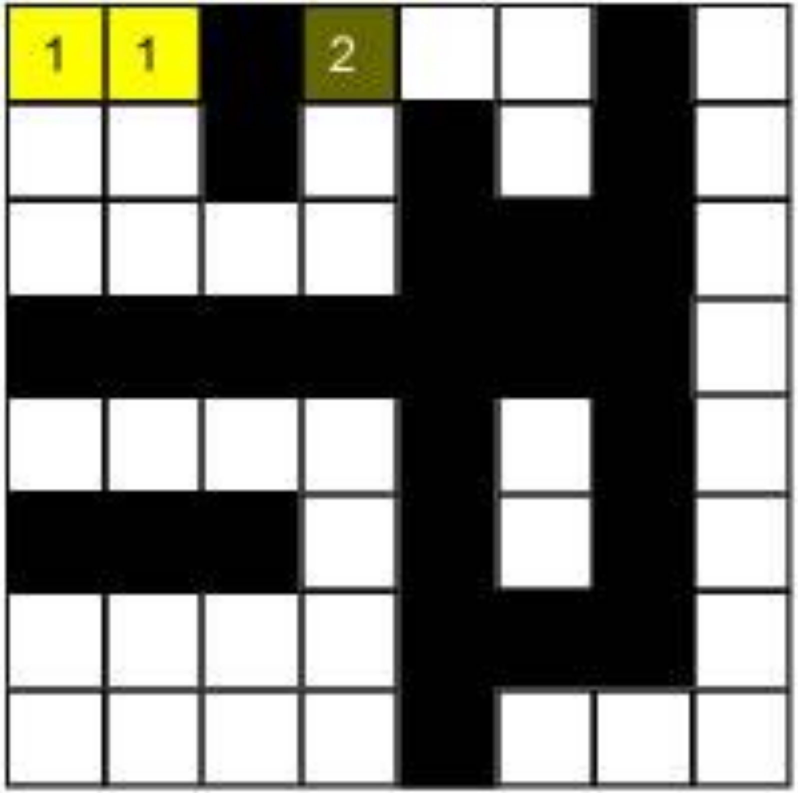


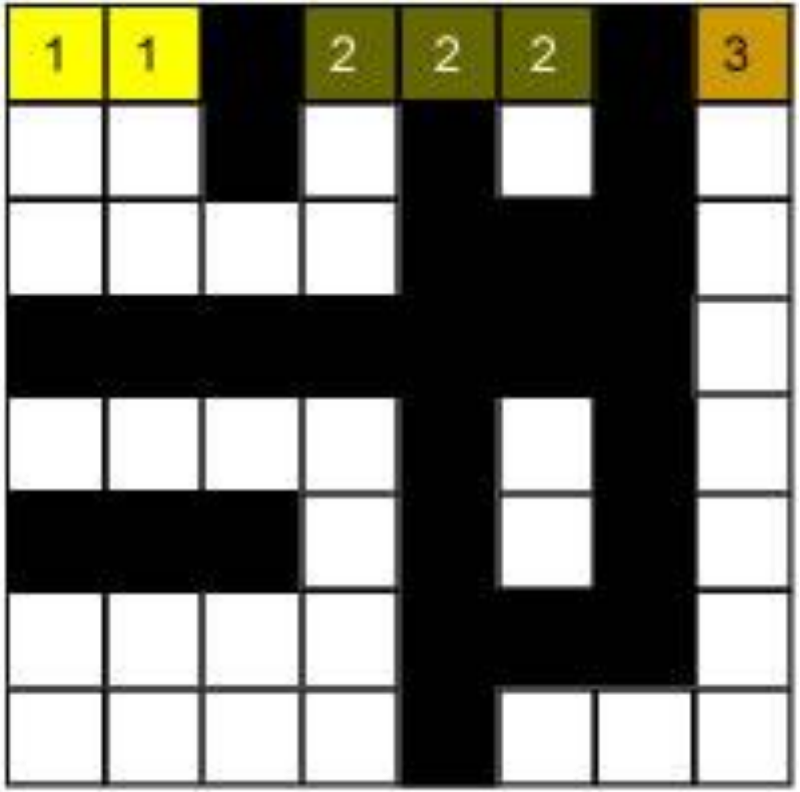
2PA: left pixel labeled→ Copy label ; left pixel BG → new label





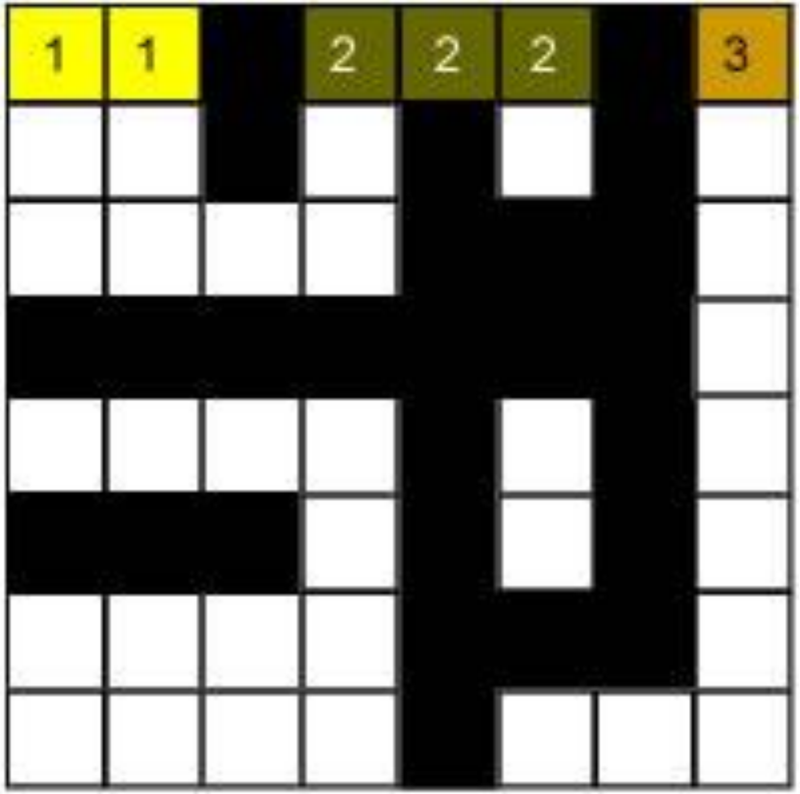
2PA: After row 1 is done

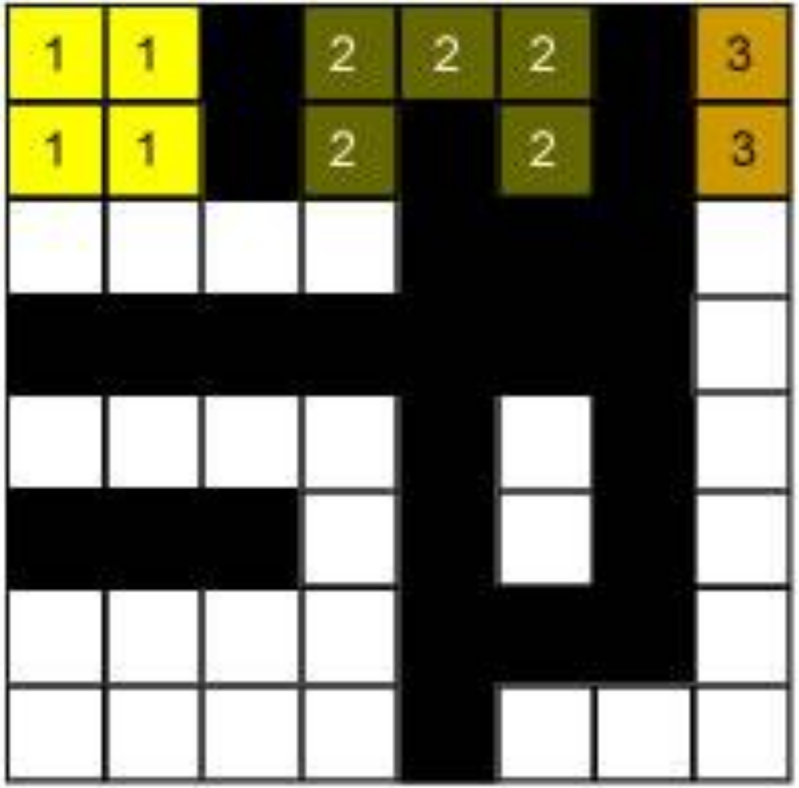




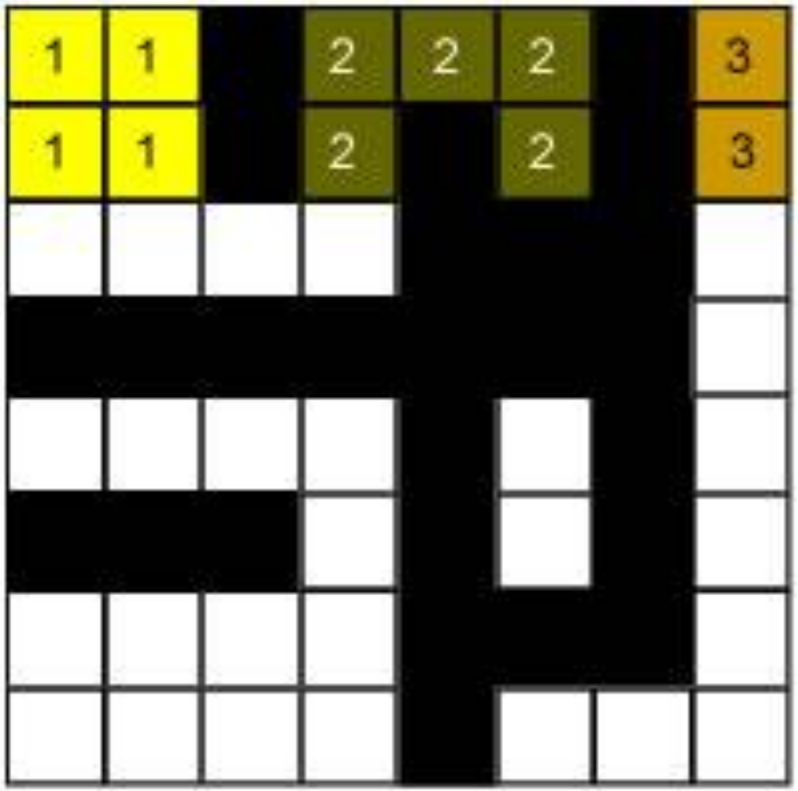
2PA: left/top pixel labeled→ Copy label

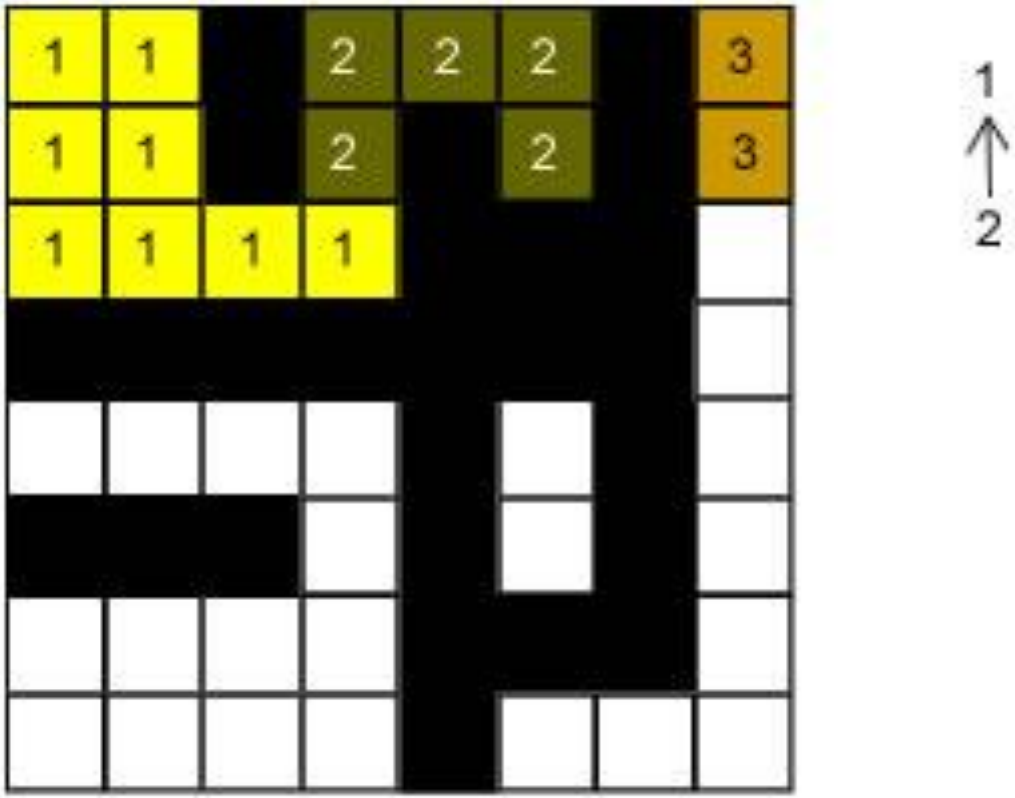
(overrides) left pixel BG → new label



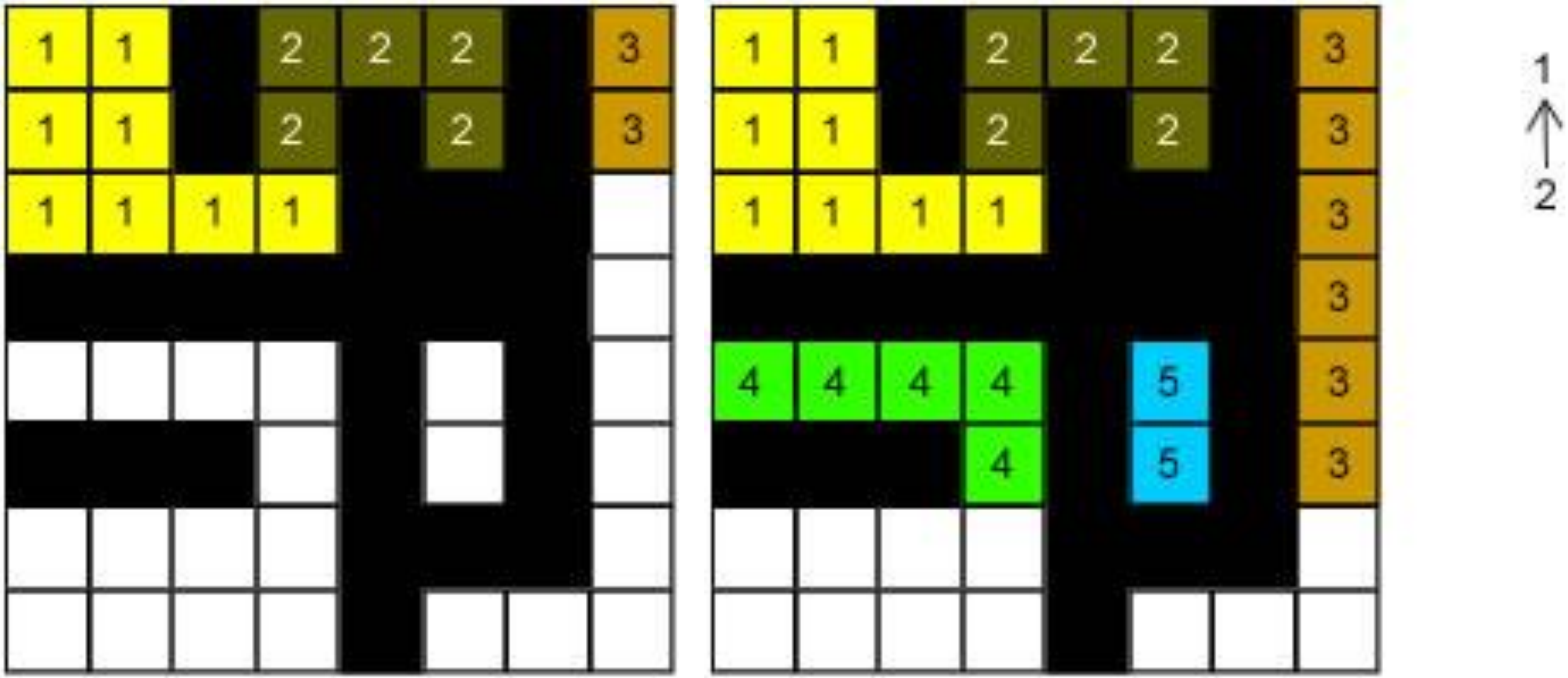


2PA: left/top pixel labeled, different labels →Copy smaller id label, record the association

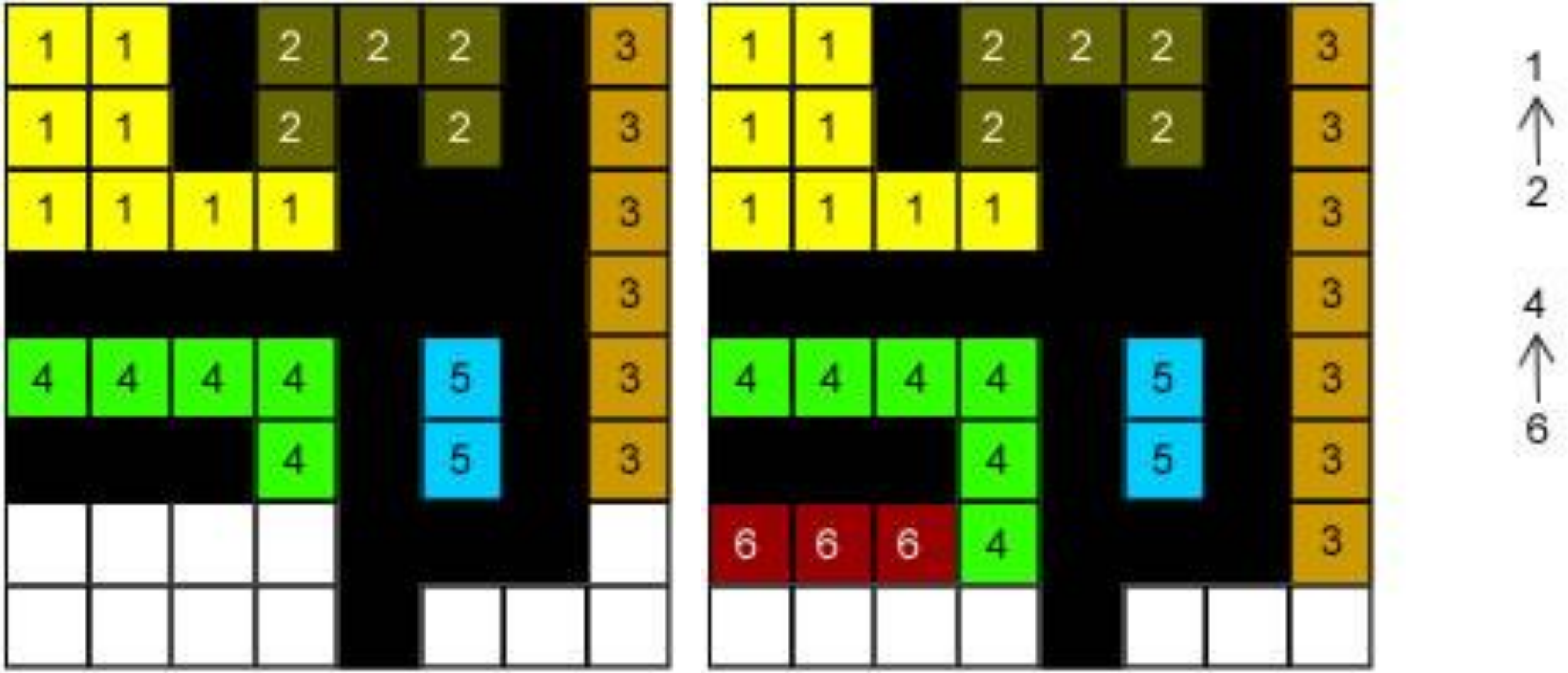




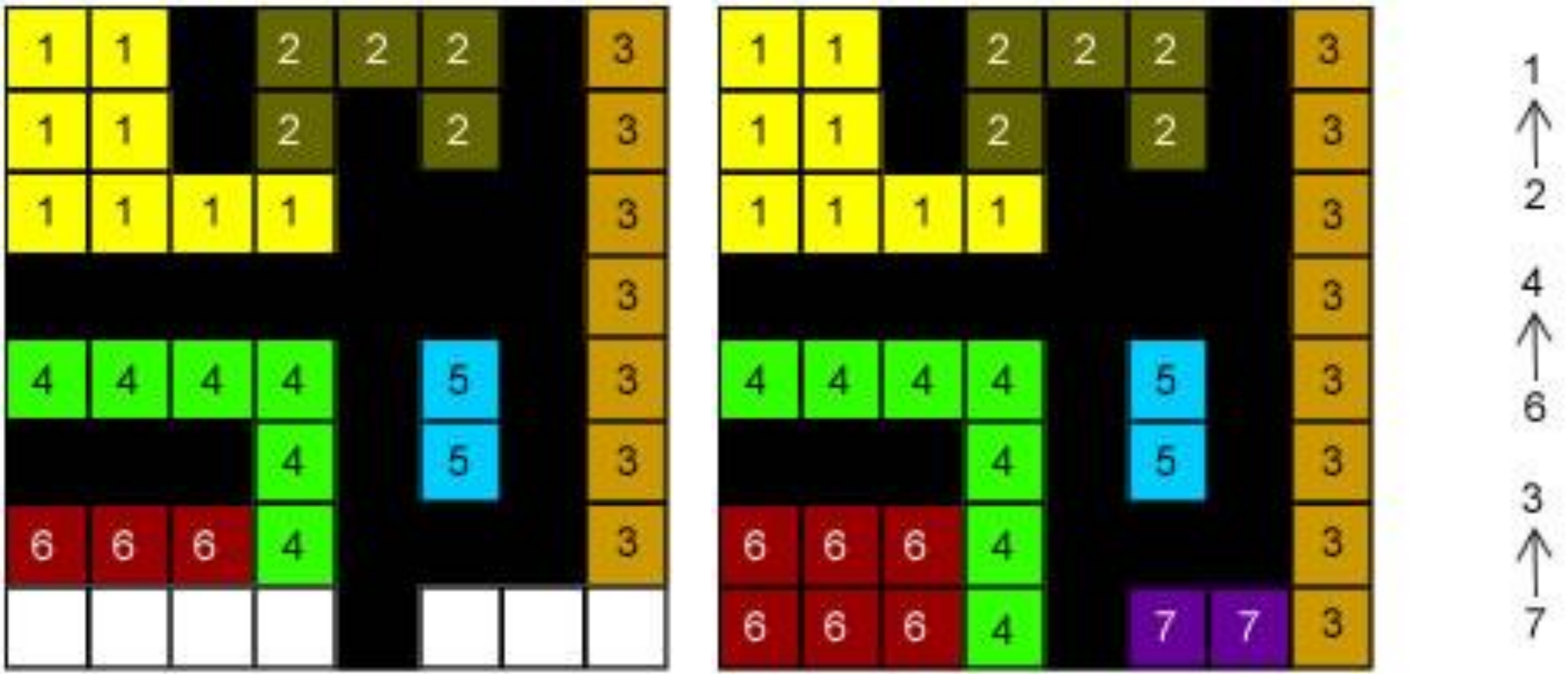
2PA: left/top pixel labeled, different labels →Copy smaller id label, record the association



2PA

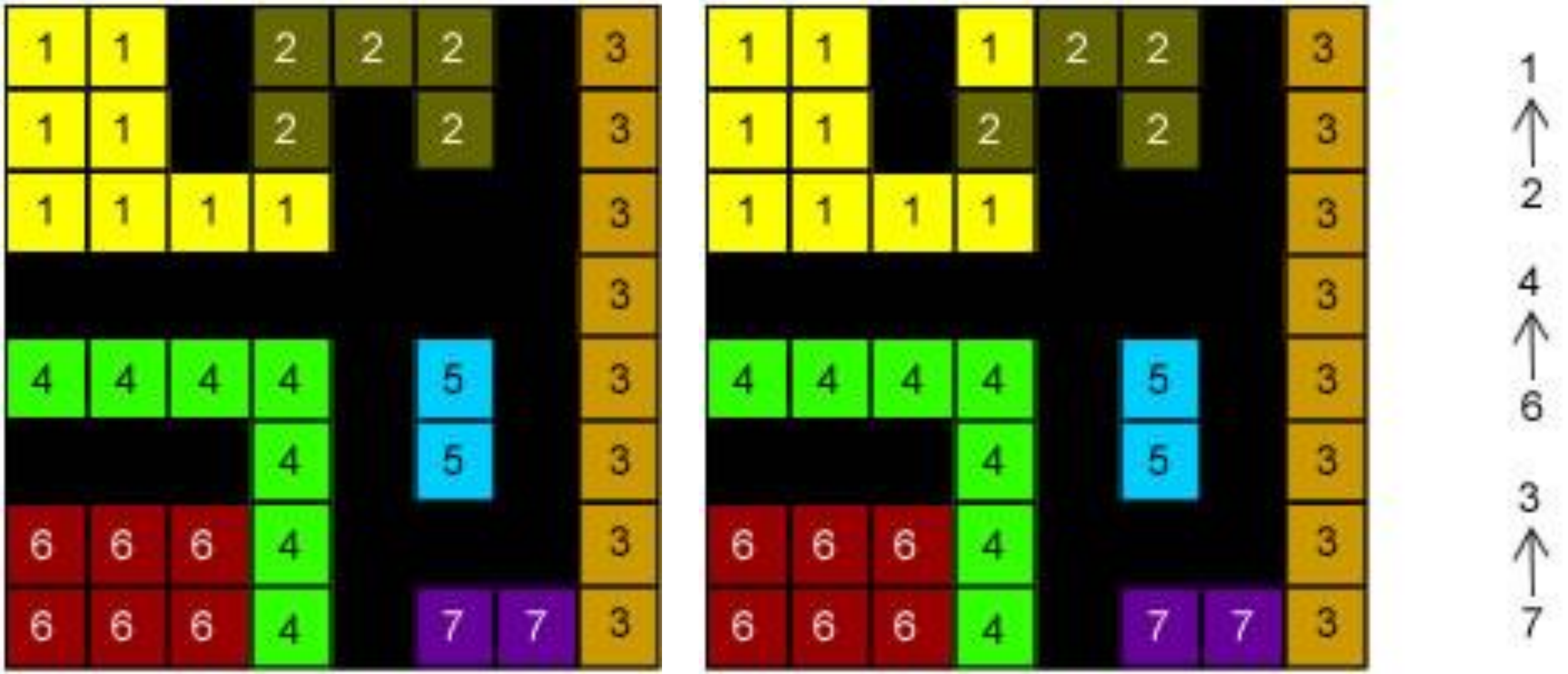


2PA: First Pass is complete



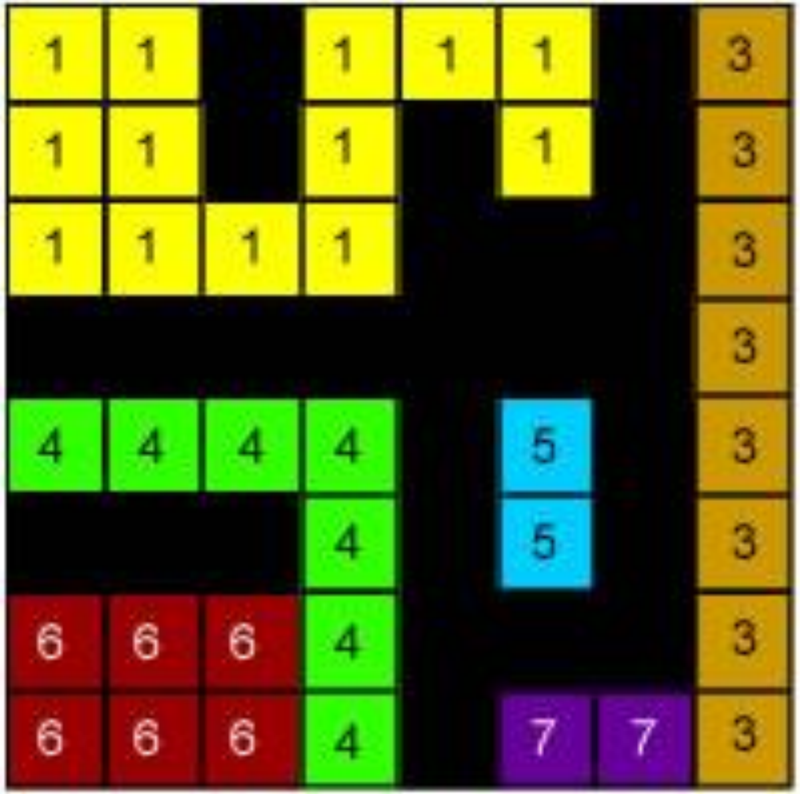
2PA: Second pass: Replace child label with root label.

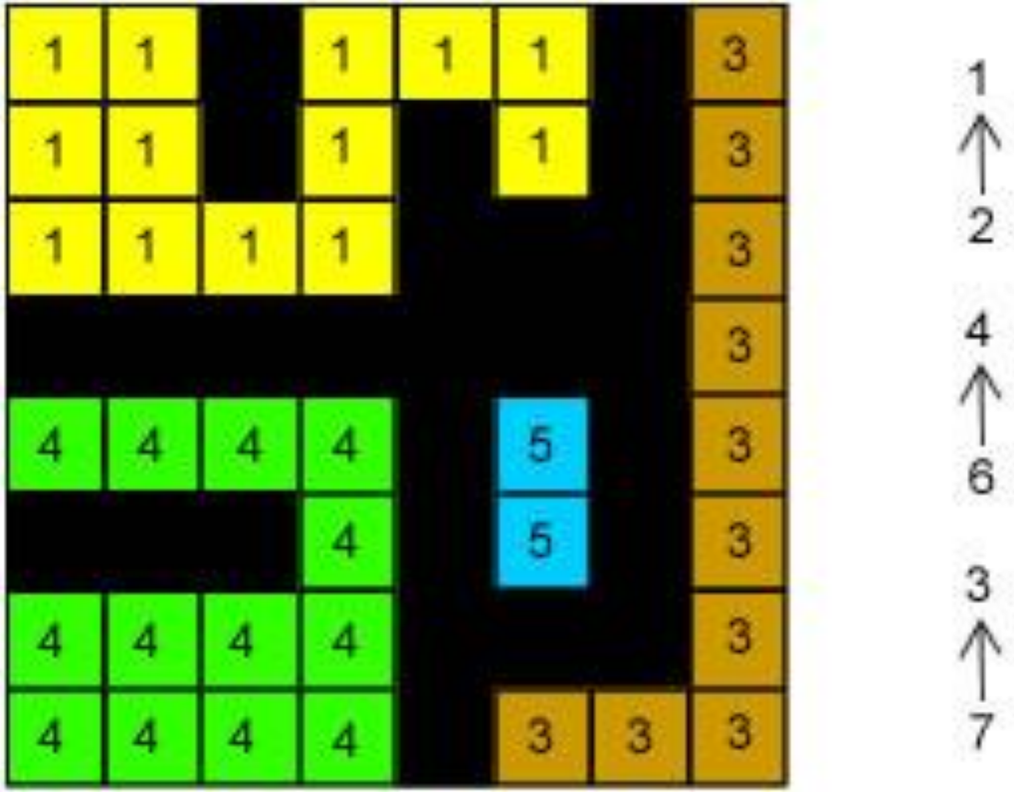
Union-Find data structure ensures ‘find’-ing is O(log*n), converges to O(1) for repeated calls.



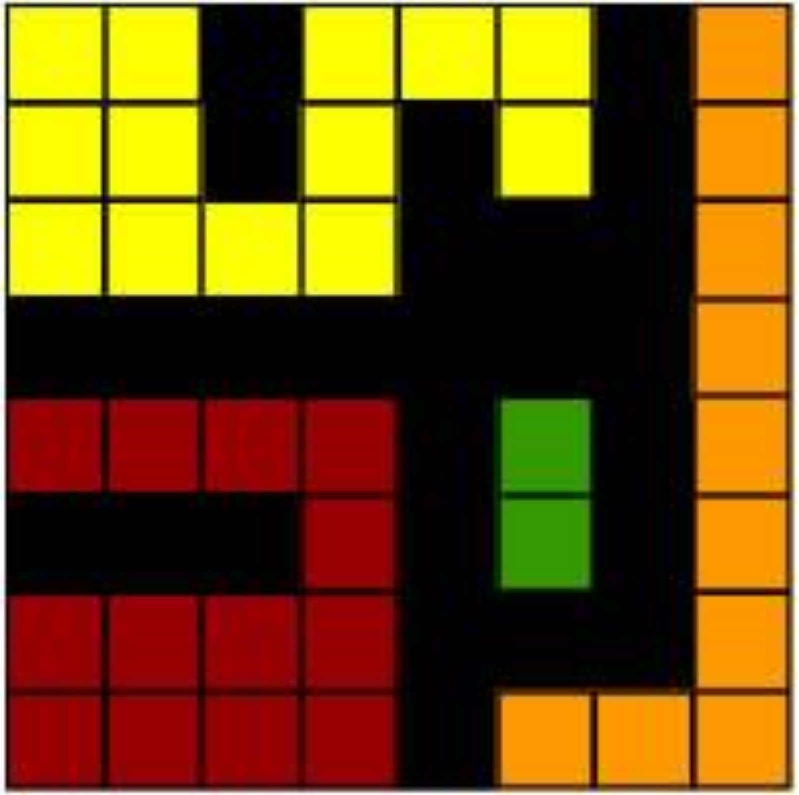
2PA: Second pass: Replace child label with root label.

Union-Find data structure ensures 'find'-ing is O(log*n), converges to O(1) for repeated calls.

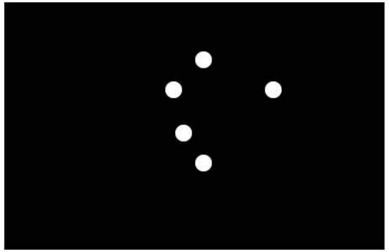




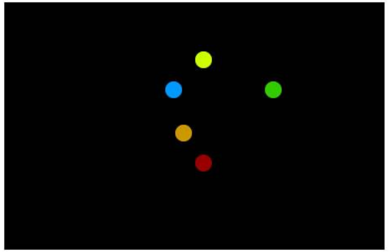
2PA: Second pass is complete



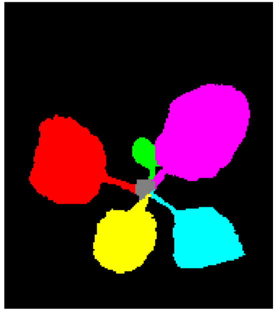
Connected Component Labeling

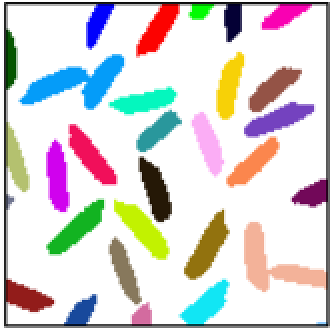


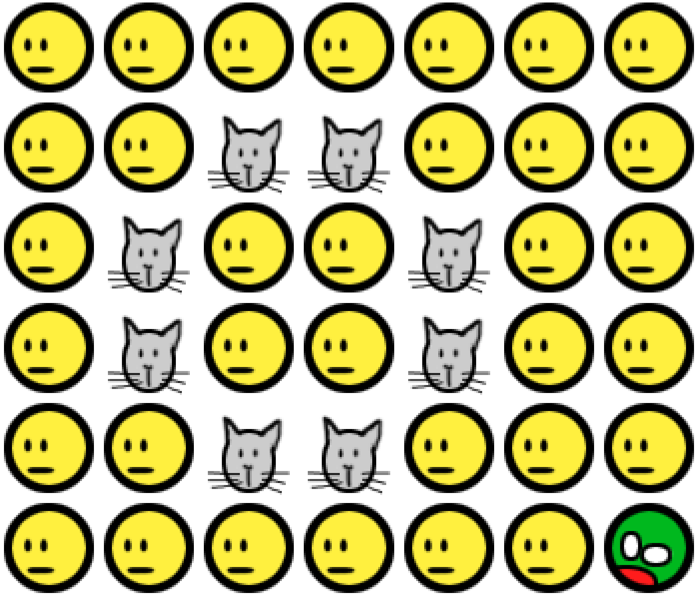
bwlabel

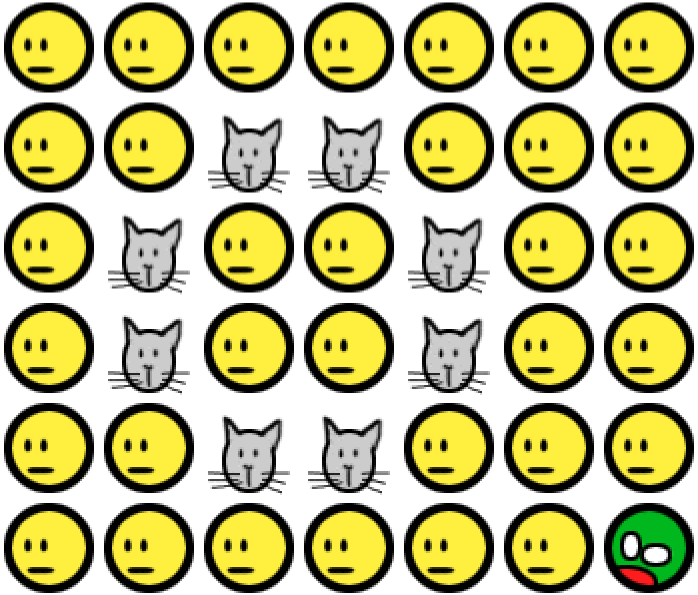


label2rgb



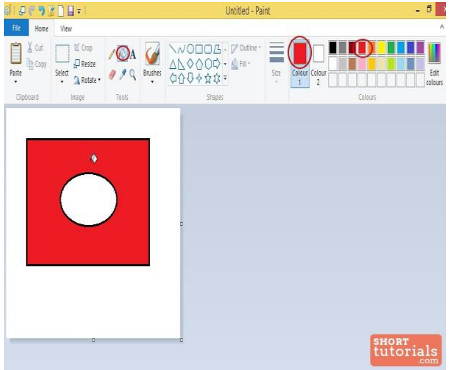


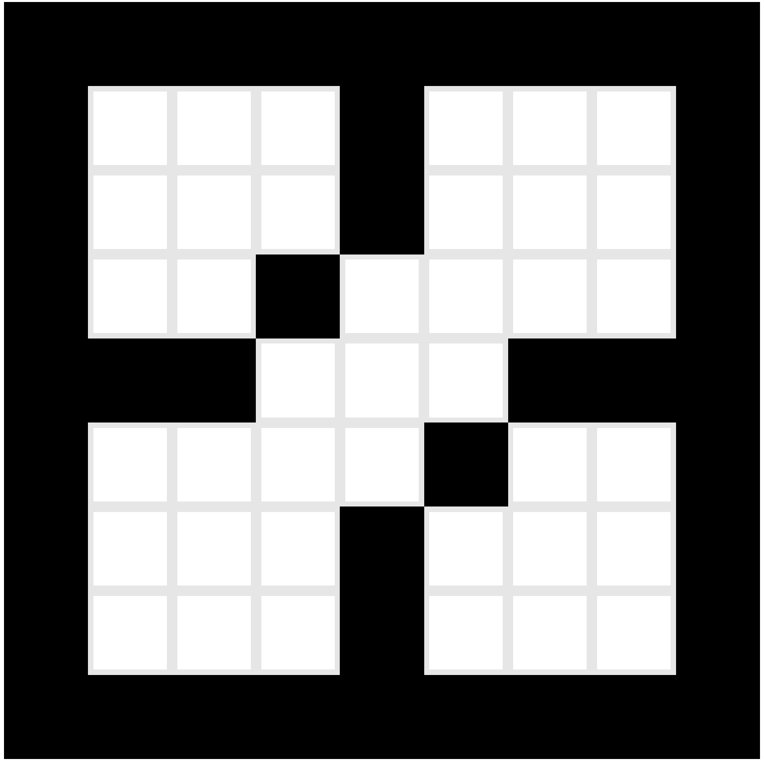




http://inventwithpython.com/blogstatic/floodfill/zombie2.gif

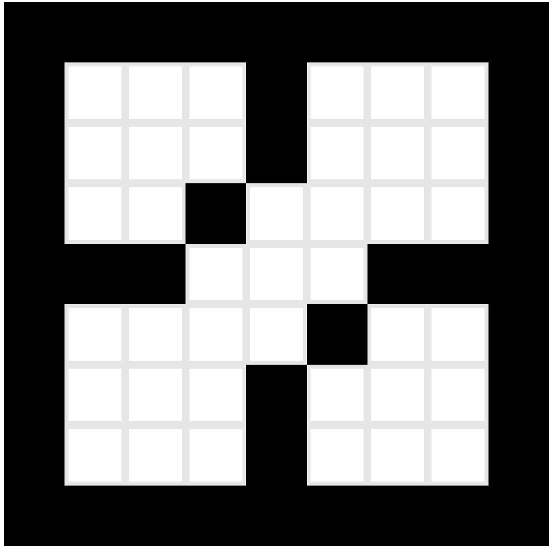
Flood-Fill





https://upload.wikimedia.org/wikipedia/commons/7/7e/Recursive_Flood_Fill_4_%28aka%29.gif

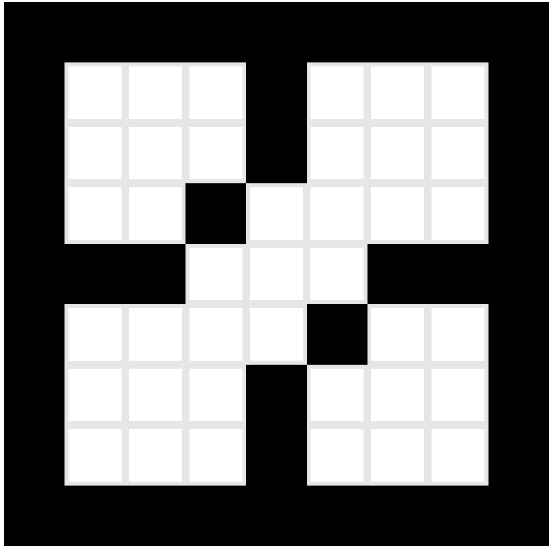
Flood-Fill Algorithm (4-conn)

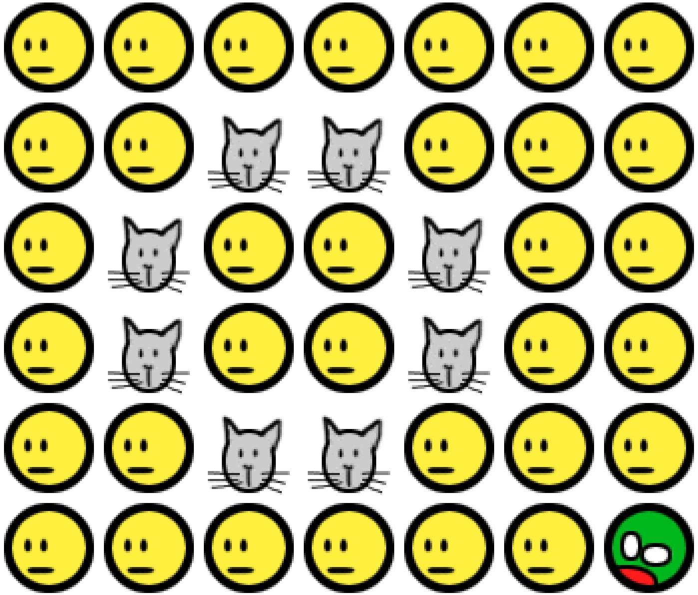


```
void floodFill(int x, int y, int fill, int old)
{
    if ((x < 0) || (x >= width)) return;
    if ((y < 0) || (y >= height)) return;
    if (getPixel(x, y) == old) {
            setPixel(fill, x, y);
            floodFill(x+1, y, fill, old);
            floodFill(x, y+1, fill, old);
            floodFill(x-1, y, fill, old);
            floodFill(x, y-1, fill, old);
    }
}
```

Many/More efficient versions exist !

If 8-conn, all white pixels filled
(all humans eventually become zombie !)





Summary of Morphological Filtering

Operation	Equation	Comments (The Roman numerals refer to the structuring elements shown in Fig. 9.26).	MATLAB codes
Translation	$(A)_z = \{w \mid w = a + z, \quad \text{for } a \in A\}$	Translates the origin of A to point z.	circshift(A,z)
Reflection	$\hat{B} = \{w \mid w = -b, \quad \text{for } b \in B\}$	Reflects all elements of B about the origin of this set.	fliplr(flipud(B))
Complement	$A^c = \{w \mid w \notin A\}$	Set of points not in A.	~A or 1-A
Difference	$A - B = \{w \mid w \in A, w \notin B\} = A \cap B^c$	Set of points that belong to A but not to B.	A &~B
Dilation	$A \oplus B = \{z \mid (\hat{B})_z \cap A \neq \varnothing\}$	"Expands" the boundary of A. (I)	imdilate(A,B)
Erosion	$A \ominus B = \{z \mid (B)_z \subseteq A\}$	"Contracts" the boundary of A. (I)	imerode(A,B)
Opening	$A \circ B = (A \ominus B) \oplus B$	Smoothes contours, breaks narrow isthmuses, and eliminates small islands and sharp peaks. (I)	imopen(A,B)
Closing	$A \bullet B = (A \oplus B) \ominus B$	Smoothes contours, fuses narrow breaks and long thin gulfs, and eliminates small holes. (I)	imclose(A,B)

Summary (Con’d)

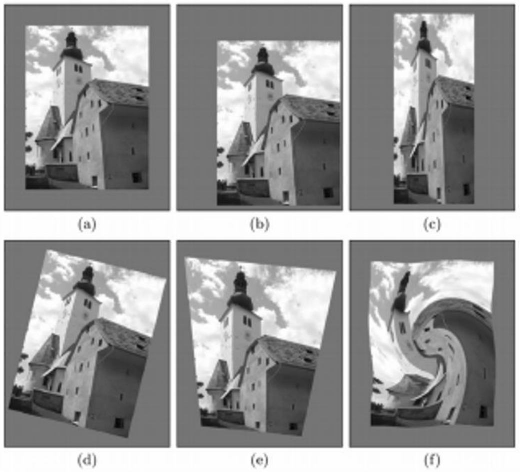
Hit-or-miss transform	$A \circledast B = (A \ominus B_1) \cap (A^c \ominus B_2)$ $= (A \ominus B_1) - (A \oplus \hat{B}_2)$	The set of points (coordinates) at which, simultaneously, B_1 found a match ("hit") in A and B_2 found a match in A^c.	bwhitmiss(A,B)
Boundary extraction	$\beta(A) = A - (A \ominus B)$	Set of points on the boundary of set A. (I)	A&~(imerode(A,B))
Region filling	$X_k = (X_{k-1} \oplus B) \cap A^c; X_0 = p$ and $k = 1, 2, 3, \ldots$	Fills a region in A, given a point p in the region. (II)	region_fill.m
Thinning	$A \otimes B = A - (A \circledast B)$ $= A \cap (A \circledast B)^c$ $A \otimes \{B\} =$ $((\ldots((A \otimes B^1) \otimes B^2) \ldots) \otimes B^n)$ $\{B\} = \{B^1, B^2, B^3, \ldots, B^n\}$	Thins set A. The first two equations give the basic definition of thinning. The last two equations denote thinning by a sequence of structuring elements. This method is normally used in practice. (IV)	bwmorph(A,’thin’);

Morphological Filtering using MATLAB

- https://in.mathworks.com/help/images/morphological-filtering.html

GEOMETRIC OPERATIONS

Geometric Operations



- Example applications of geometric operations:
  - Zooming images, windows to arbitrary size
  - Computer graphics: deform textures and map to arbitrary surfaces
- **Definition:** Geometric operation transforms image *I* to new image *I'* by modifying **coordinates of image pixels**

$$I(x, y) \rightarrow I'(x', y')$$

- Intensity value originally at (x,y) moved to new position (x',y')

(x,y)

(x + d_x, y + d_y)

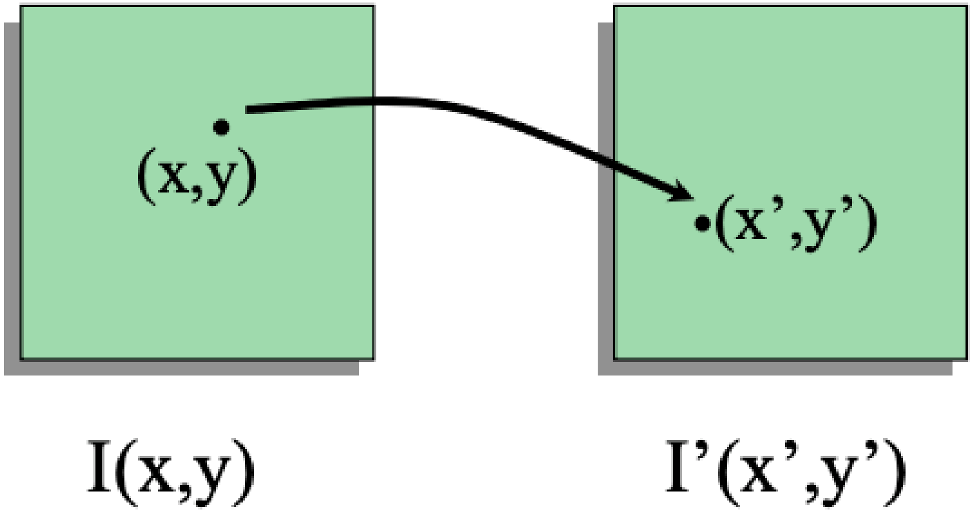
Example: Translation geometric operation moves value at (x,y) to (x + d_x, y + d_y)

https://web.cs.wpi.edu/~emmanuel/courses/cs545/S14/slides/lecture11.pdf

$$x \rightarrow f_x(x, y) = x'$$

$$y \rightarrow f_y(x, y) = y'$$

$$I(x, y) = I'\left(f_x(x, y), f_y(x, y)\right)$$

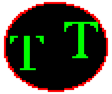


Common Geometric Operations

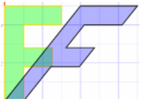
- **Scale** - change image content size

- **Rotate** - change image content orientation

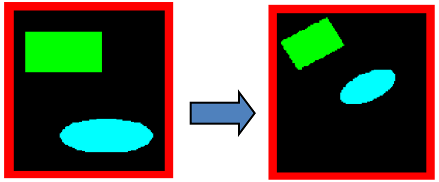
- **Reflect** - flip over image contents



- **Translate** - change image content position



- **Shear** – shift points along a line parallel to fixed line



- **Affine Transformation**
  - general image content linear geometric transformation

Simple Mappings

- **Translation:** (shift) by a vector (d_x, d_y)

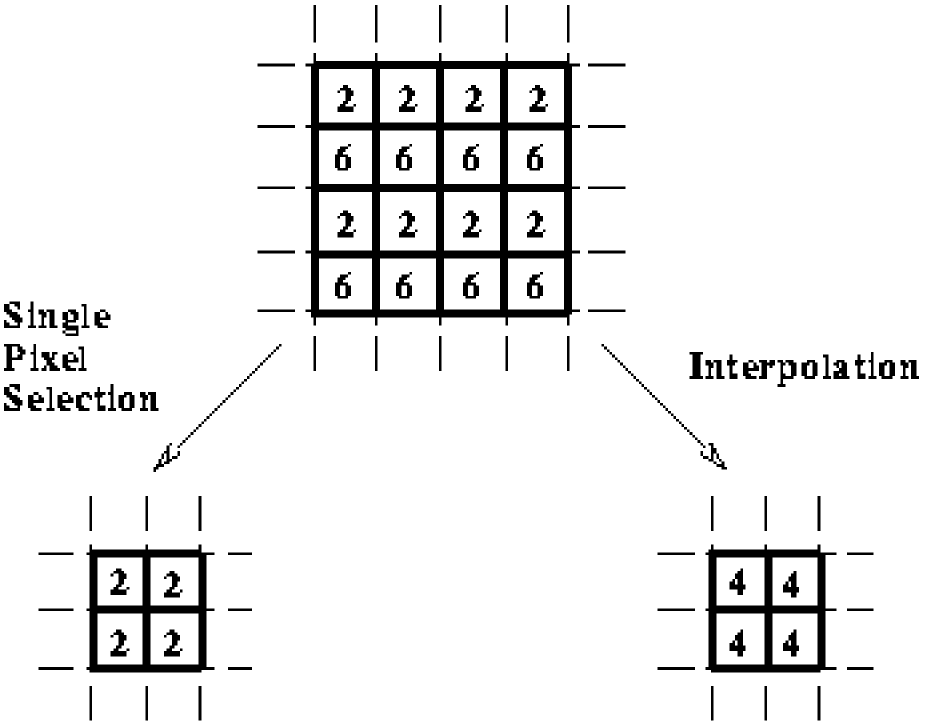
$$T_x : x' = x + d_x \qquad T_y : y' = y + d_y \qquad \text{or} \qquad \begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} d_x \\ d_y \end{pmatrix}$$

- **Scaling:** (contracting or stretching) along x or y axis by a factor s_x or s_y

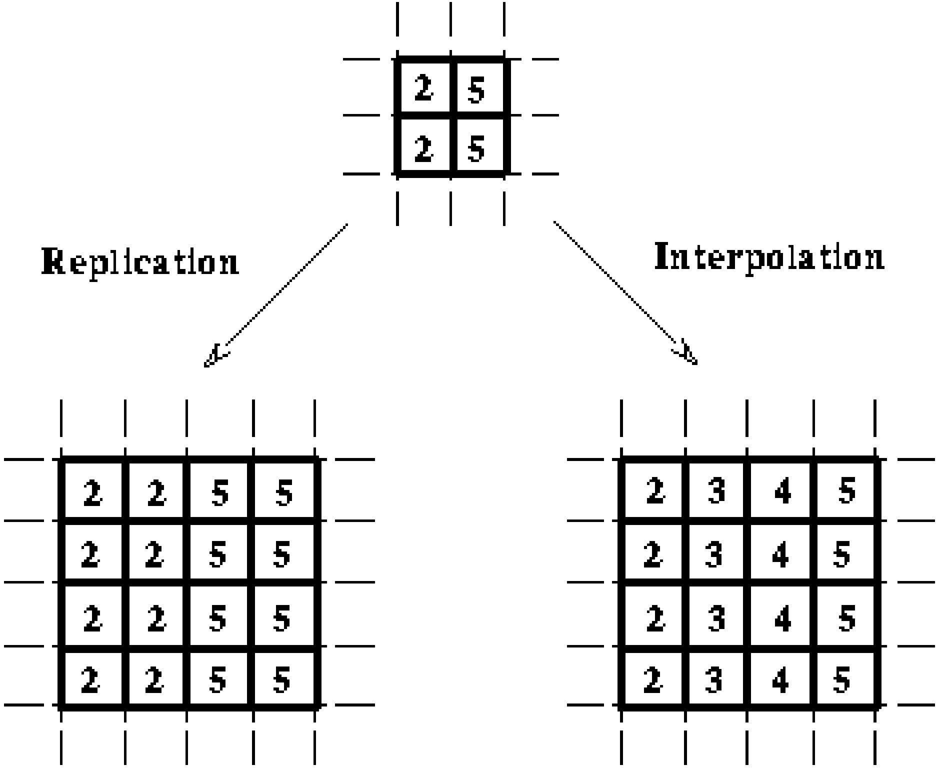
$$T_x : x' = s_x \cdot x \qquad T_y : y' = s_y \cdot y \qquad \text{or} \qquad \begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} s_x & 0 \\ 0 & s_y \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix}$$

https://web.cs.wpi.edu/~emmanuel/courses/cs545/S14/slides/lecture11.pdf

Scaling (Shrink)



Scaling (Stretch)



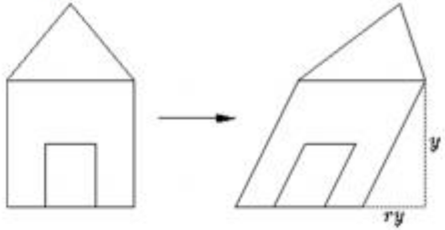
Simple Mappings

- **Shearing:** along x and y axis by factor b_x and b_y

$$T_x : x' = x + b_x \cdot y$$
$$T_y : y' = y + b_y \cdot x$$

or

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} 1 & b_x \\ b_y & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix}$$



- **Rotation:** the image by an angle α

$$T_x : x' = x \cdot \cos\alpha - y \cdot \sin\alpha$$
$$T_y : y' = x \cdot \sin\alpha + y \cdot \cos\alpha$$

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix}$$

https://web.cs.wpi.edu/~emmanuel/courses/cs545/S14/slides/lecture11.pdf

Homogeneous Coordinates

- Notation useful for converting scaling, translation, rotating into point-matrix multiplication
- To convert ordinary coordinates into homogeneous coordinates

$$\boldsymbol{x} = \begin{pmatrix} x \\ y \end{pmatrix} \quad \text{converts to} \quad \hat{\boldsymbol{x}} = \begin{pmatrix} \hat{x} \\ \hat{y} \\ h \end{pmatrix} = \begin{pmatrix} h\,x \\ h\,y \\ h \end{pmatrix}$$

Homogeneous coordinates

- Homogeneous coordinates transforms a point from Euclidean plane to projective plane by adding a dummy variable (x,y,1)
- Overall scaling is unimportant so (x,y,1) ≅ (ax,ay,a), for any non-zero ‘a’
- Because scaling does not play role (aX,aY,AW) are called homogeneous coordinates of the point
- 2 classes of transformations:
  - Projective
  - Affine (special case of projective, c1=c2=0,

$$\begin{pmatrix} a_1 & a_2 & b_1 \\ a_3 & a_4 & b_2 \\ c_1 & c_2 & 1 \end{pmatrix} \times \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix}$$

All linear transformations are affine but not all affine transformations are linear

http://robotics.stanford.edu/~birch/projective/node4.html
https://www.graphicsmill.com/docs/gm5/Transformations.htm

Affine (3-Point) Mapping

Translation T'

Counter-clockwise rotation θ

Scaling s

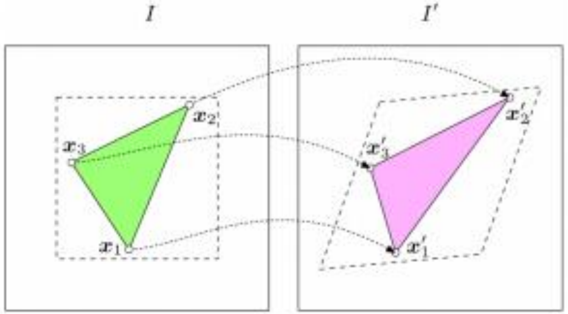
Another translation T

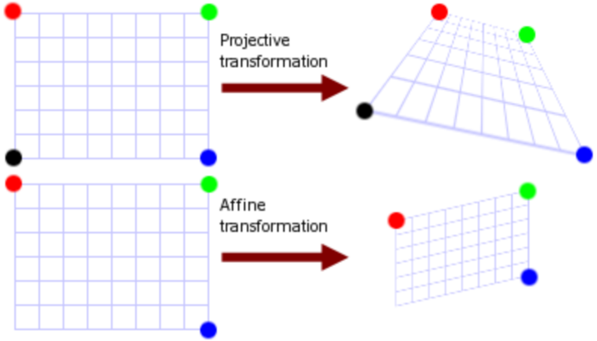
$$\begin{bmatrix} s_x \cos\theta & -s_y \sin\theta & t_x s_x \cos\theta - t_y s_y \sin\theta + t'_x \\ s_x \sin\theta & s_y \cos\theta & t_x s_x \sin\theta + t_y s_y \cos\theta + t'_y \\ 0 & 0 & 1 \end{bmatrix}$$

- Can use homogeneous coordinates to rewrite translation, rotation, scaling, etc as vector-matrix multiplication

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

- **Affine mapping:** Can then derive values of matrix that achieve desired transformation (or combination of transformations)





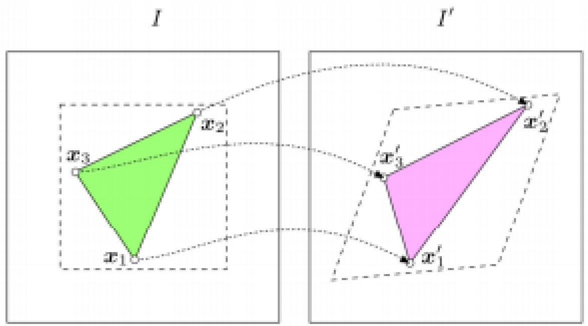
- Inverse of transform matrix is **inverse mapping**

https://web.cs.wpi.edu/~emmanuel/courses/cs545/S14/slides/lecture11.pdf

https://www.graphicsmill.com/docs/gm5/Transformations.htm

Affine (3-Point) Mapping

- What's so special about affine mapping?



- Maps
  - straight lines -> straight lines,
  - triangles -> triangles
  - rectangles -> parallelograms
  - Parallel lines -> parallel lines
- Distance ratio on lines do not change

References

- G&W, 3rd Ed., 9.1-9.3, 9.6
- Grayscale Morphology: http://people.ciirc.cvut.cz/~hlavac/TeachPresEn/11ImageProc/71-06MatMorfolGrayEn.pdf