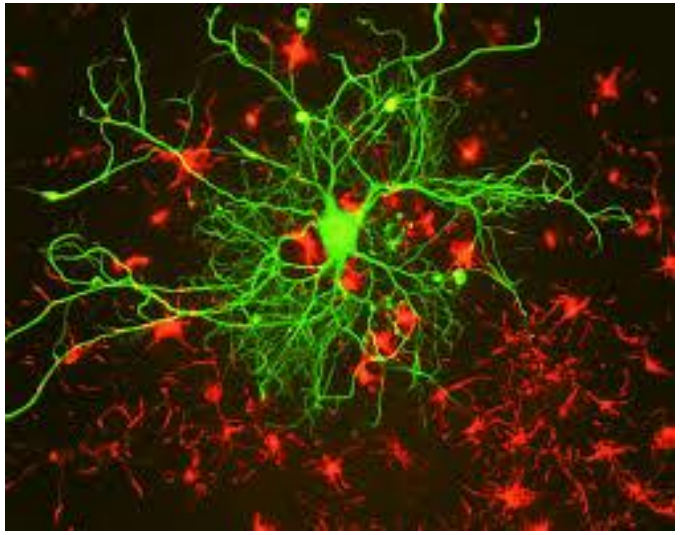
Announcement

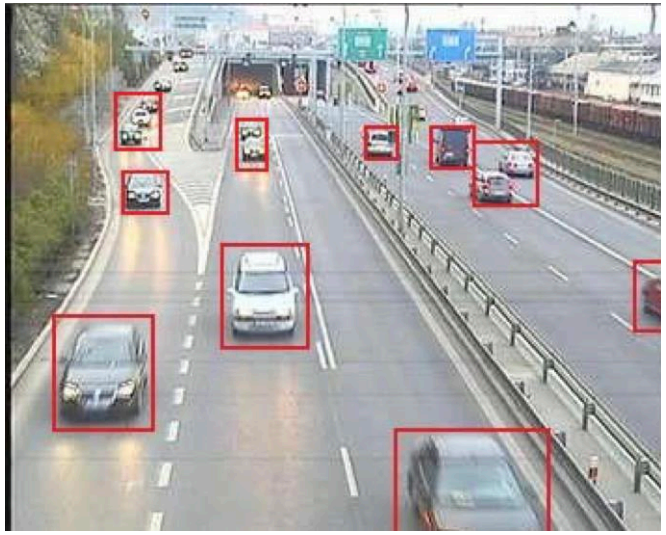
2/18/2025 Tuesday

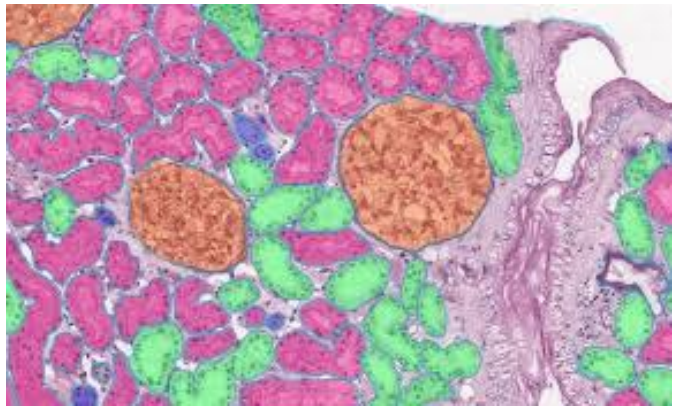
- Image segmentation
  - Motivation
  - Category
  - Algorithms

Motivation

- Segmentation is prerequisite to downstream quantitative analysis tasks

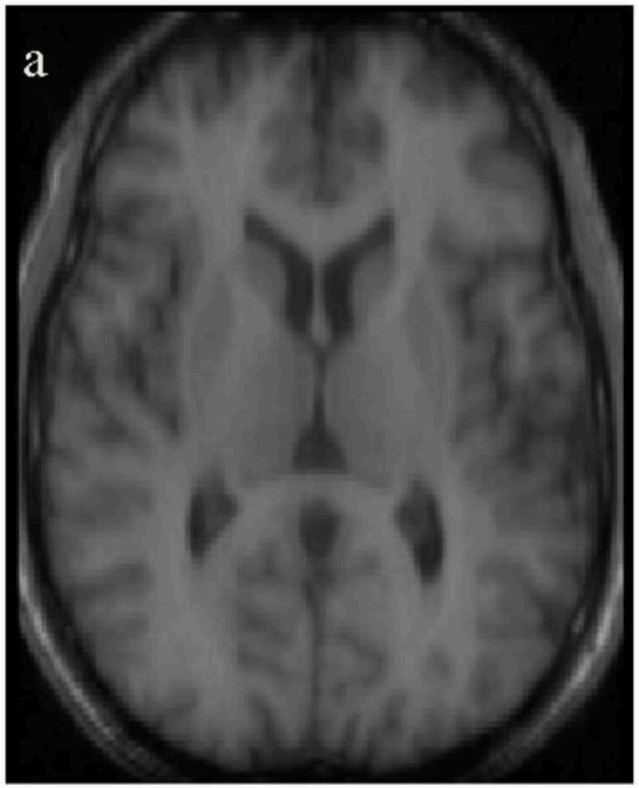






Segmentation task

- Partition a region into subregions and assign a label to each pixel



Input image

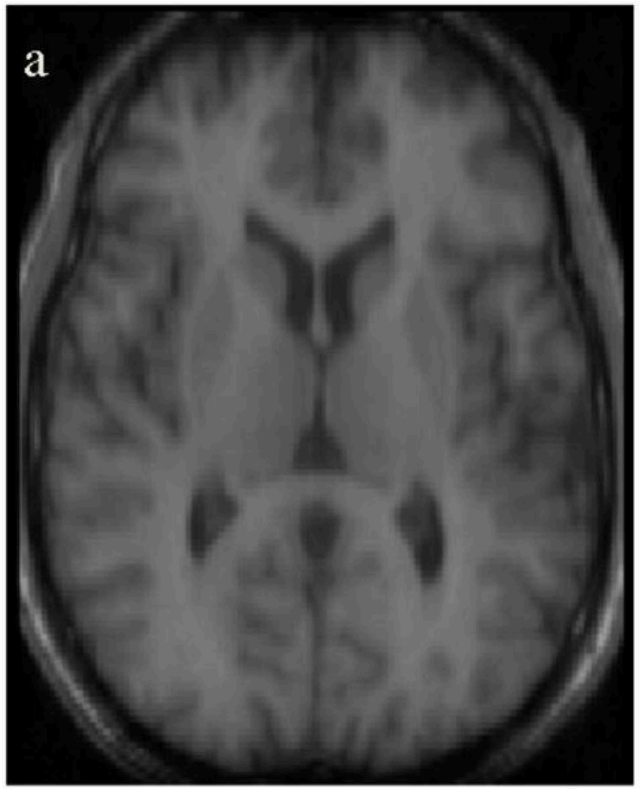
For visualization, assign a color per label

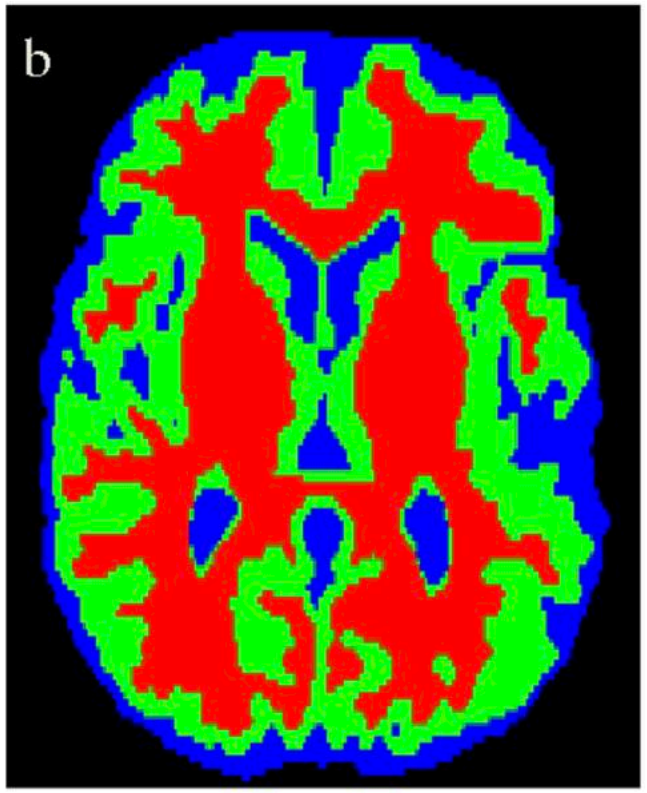
2	2	2	2	2	1	1	1
2	2	2	2	2	1	1	1
2	1	1	1	1	1	1	1
2	1	1	1	1	1	1	4
2	1	1	1	1	1	4	4
3	3	3	1	4	4	4	4
3	3	3	3	3	4	4	4
3	3	3	3	3	4	4	4

Segmentation map

Type I: semantic segmentation

- Classify each pixel in an image into a specific category

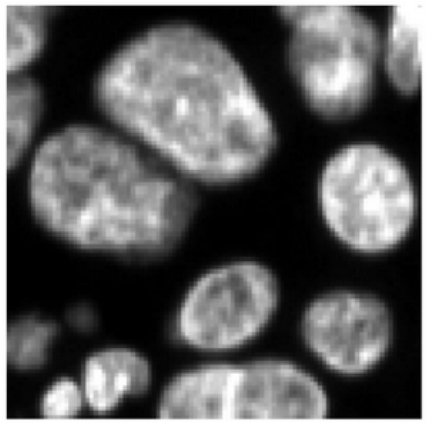




0: background
1: white matter
2: gray matter
3: fluid

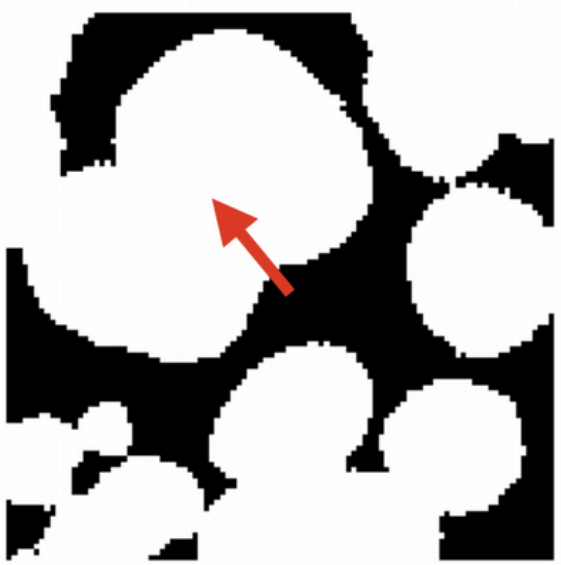
Semantic error: false positive (FP) and false negative (FN)

- L2 works: $\|\text{pred} - \text{gt}\|^2$



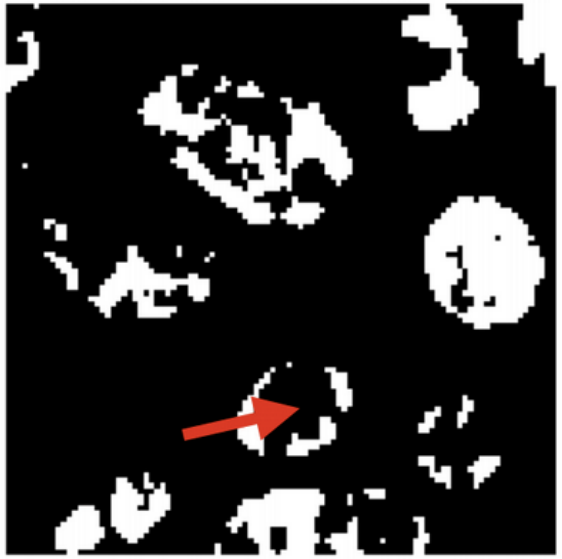
Image

Small threshold: extra pixels



False positive

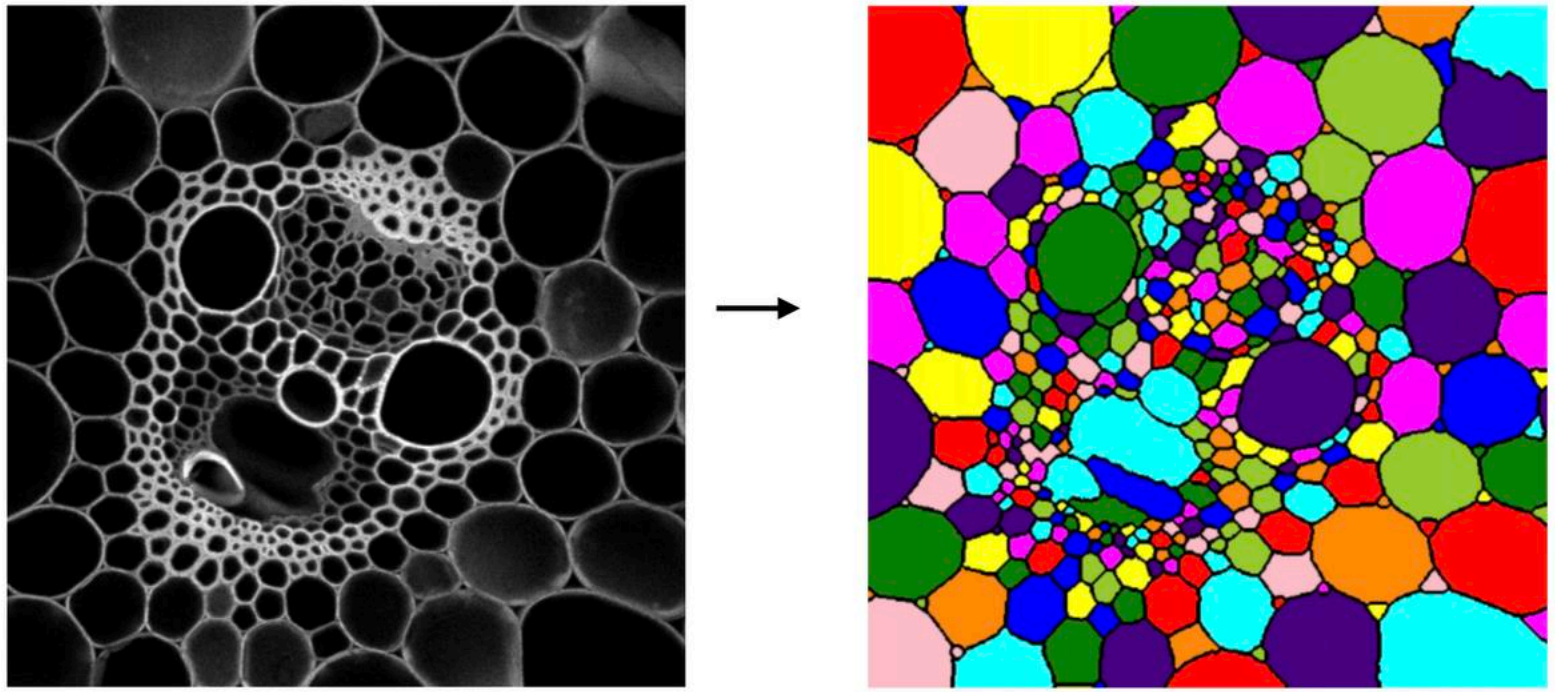
Big threshold: missing pixels



False negative

Type II: instance segmentation

- Identify and segment each individual object in an image
  - distinguish between different instances of the same object category

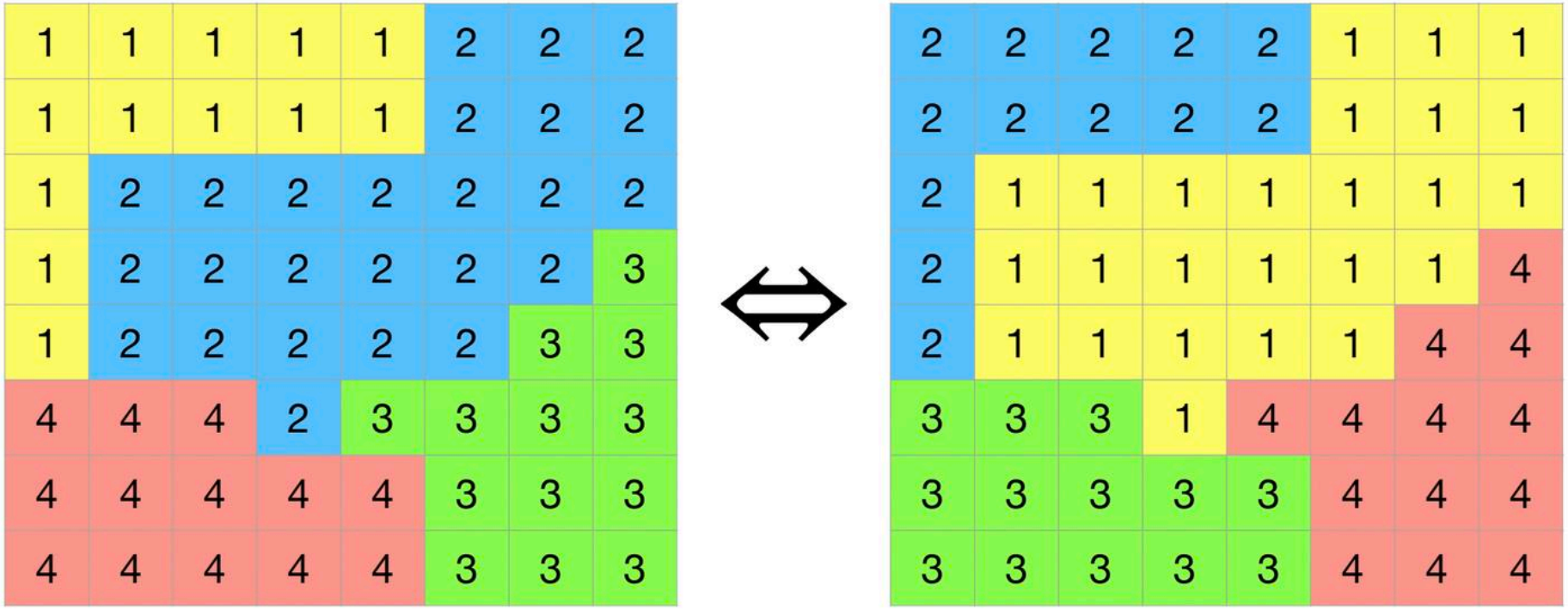


Input Image

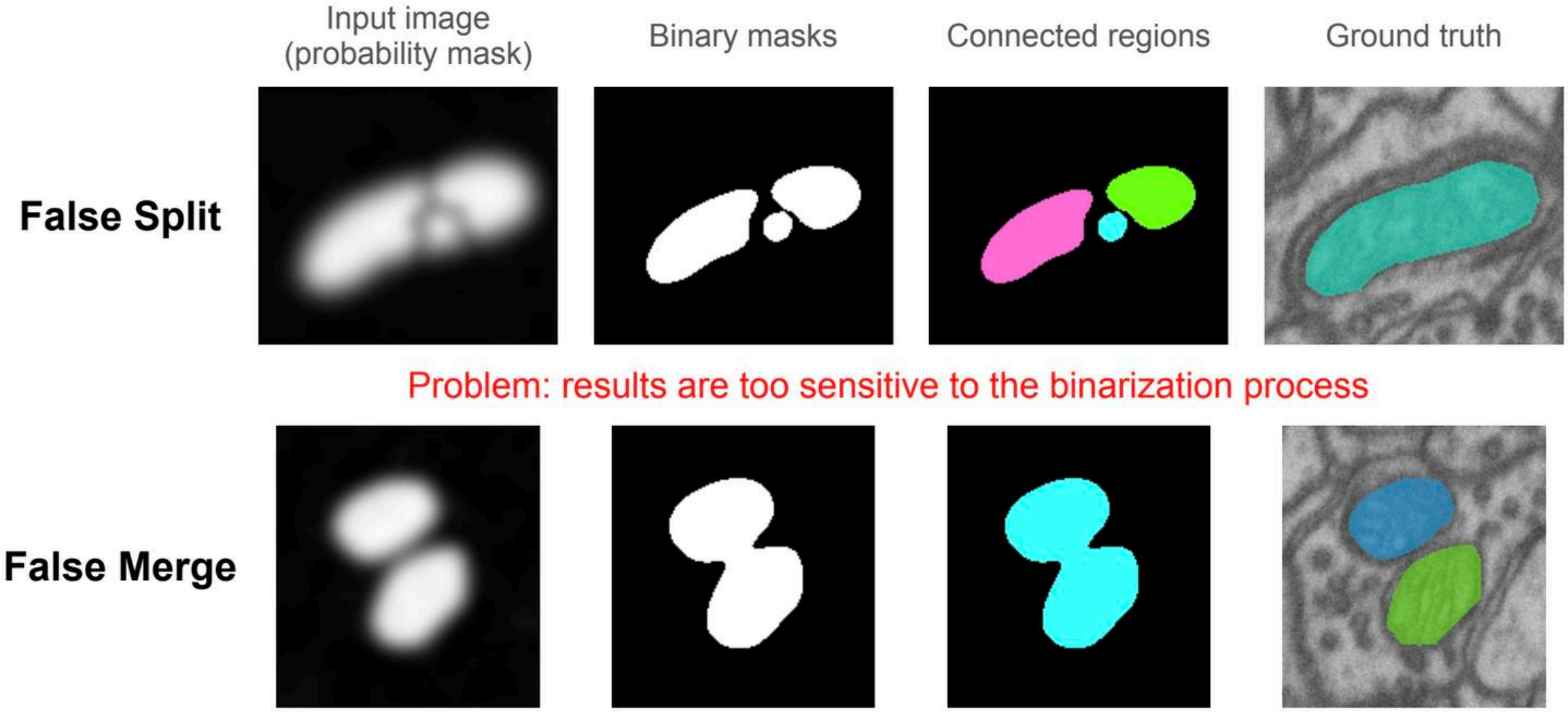
Which cell

Instance error: permutation invariance

- L2 does not work: $\|\mathrm{pred} - \mathrm{gt}\|^2$
- The group ids do not have a fixed order



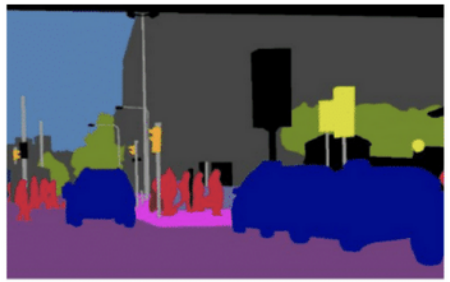
Instance error: false split and false merge



Type III: panoptic segmentation

- A unified approach that combines both semantic segmentation and instance segmentation into a single task
  - assign a category label and an instance ID to every pixel in an image

Input image



Semantic segmentation
(Stuff)

Instance segmentation
(Object)

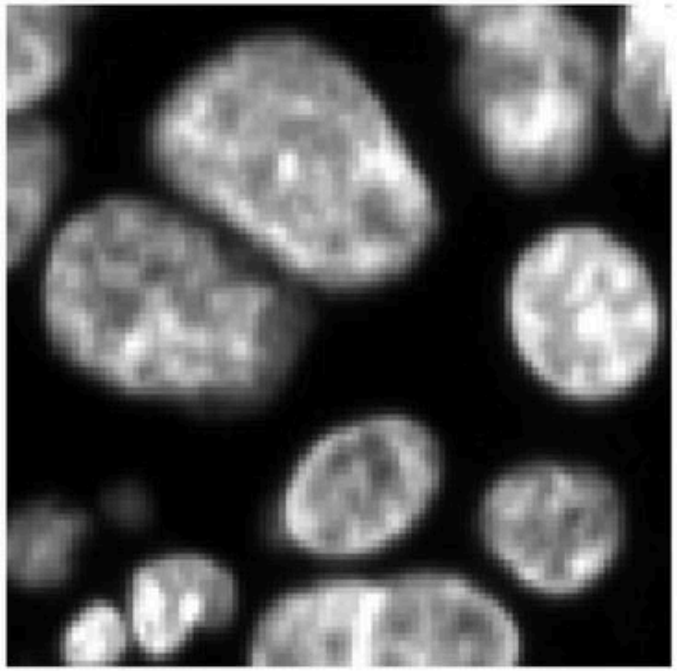
Stuff
- ❖ amorphous regions, e.g., sky, road
- ❖ not instance-specific but still segmented

Panoptic segmentation

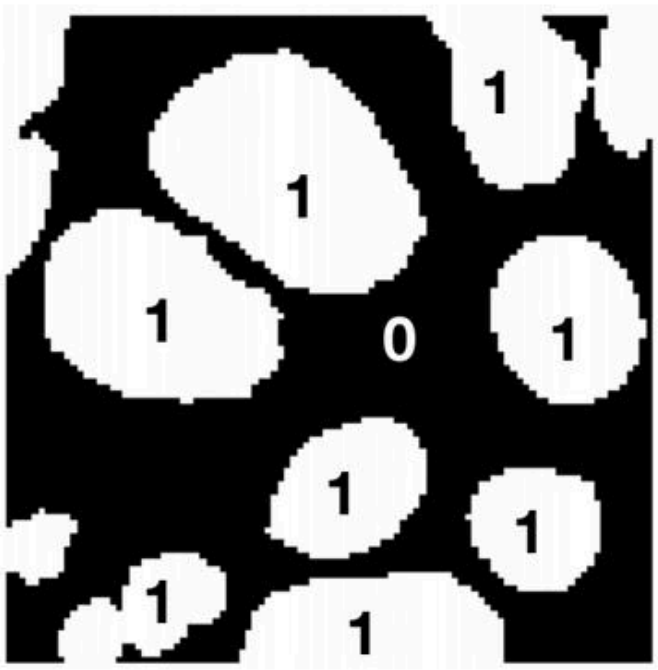
e.g., "Car 1," "Car 2," "Sky," "Road"

Thing/object
- ❖ countable objects, e.g., person, car
- ❖ segmented as individual instances

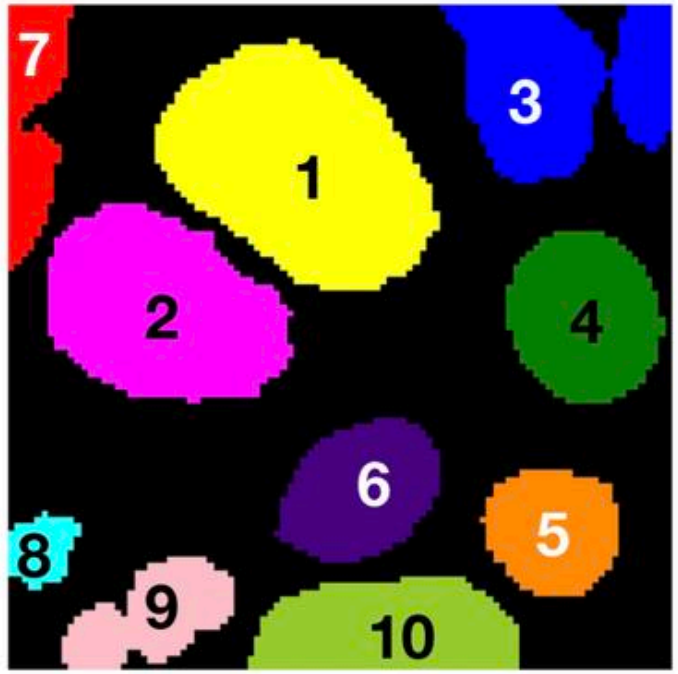
How to do semantic & instance segmentation



Image



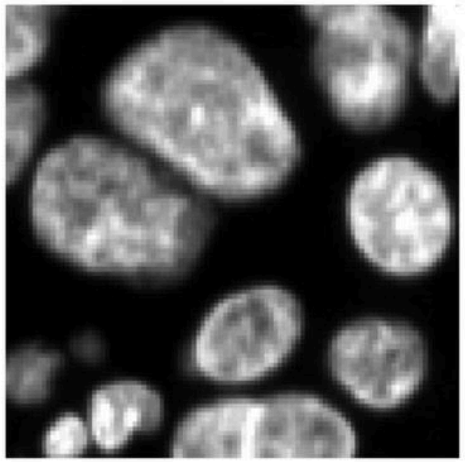
Semantic segmentation



Instance segmentation

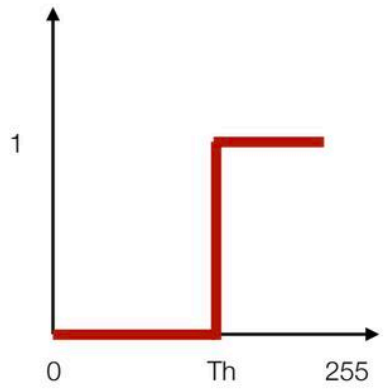
Task 1: semantic segmentation

- Baseline: thresholding



Image

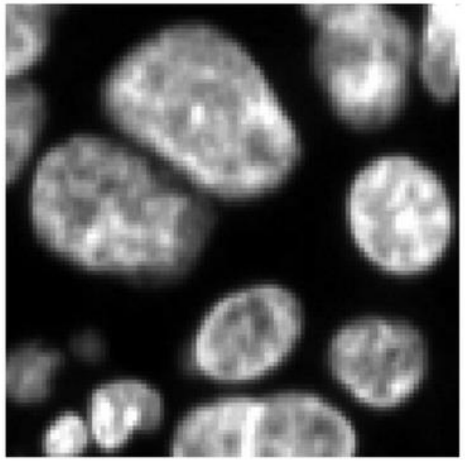
```
I[I >= Th] = 1
I[I < Th]  = 0
```



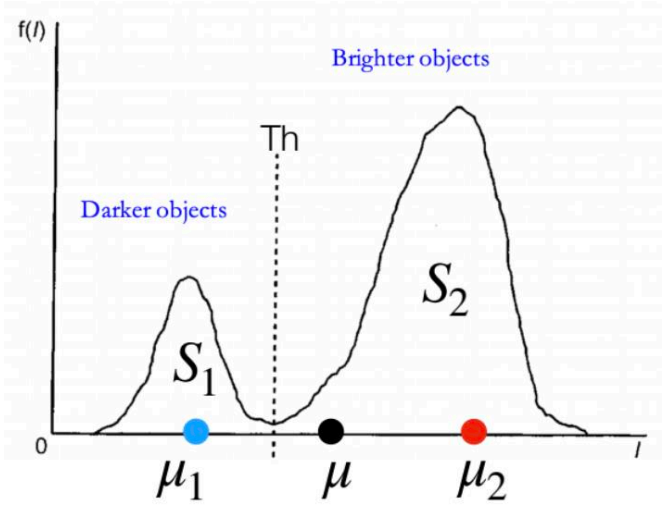
Task 1: semantic segmentation

- Automatic determination: Ostu thresholding

```
th_otsu, img1_otsu = cv2.threshold(img1, 0, 255, cv2.THRESH_BINARY+cv2.THRESH_OTSU)
```



Image



1. What is the ideal threshold?

$$\sigma_b^2 = \frac{N_1}{N}(\mu_1 - \mu)^2 + \frac{N_2}{N}(\mu_2 - \mu)^2$$

Goal: maximize between-class variance

2. How to find it?

```
mu = I.mean(); N = I.size
```

For uint8 image, do a for-loop of Th={0,..,255}

```
S1 = I >= Th
S2 = I < Th

mu1 = I[S1].mean()
mu2 = I[S2].mean()
```

```
sigma_b = (S1.sum()/N)*(mu1-mu)**2 + (S2.sum()/N)*(mu2-mu)**2
```

Task 1: semantic segmentation

- Problem: what about non-uniform lighting

Image

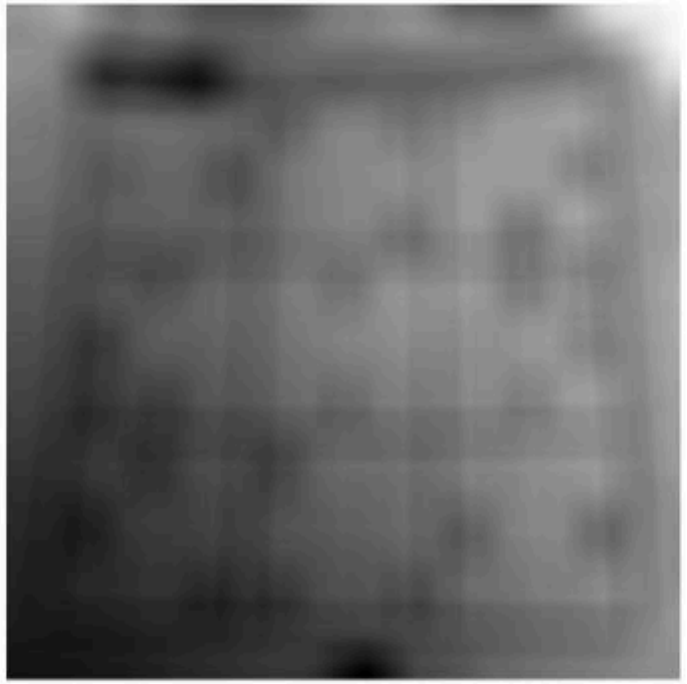
Image>127

Image>60

Task 1: semantic segmentation

- Adaptive thresholding

Image



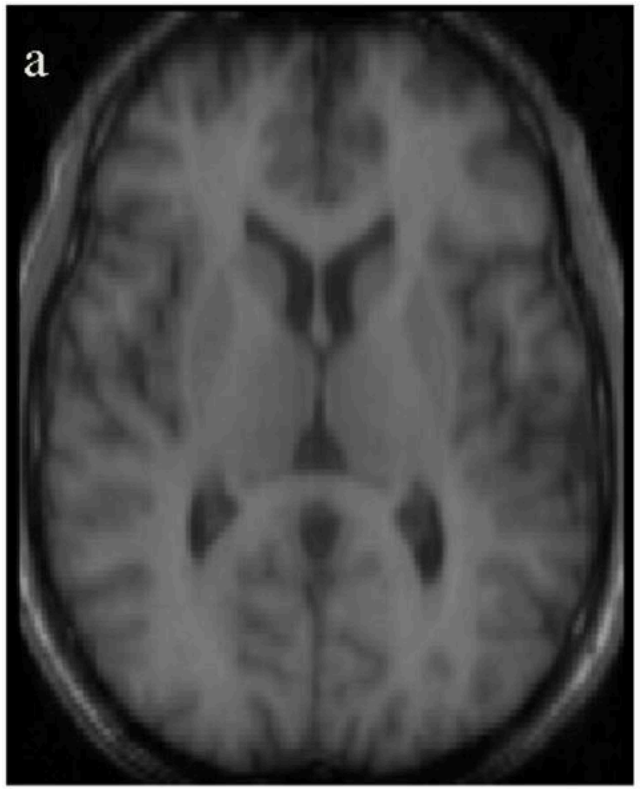
Estimated brightness
(Gaussian filter result)

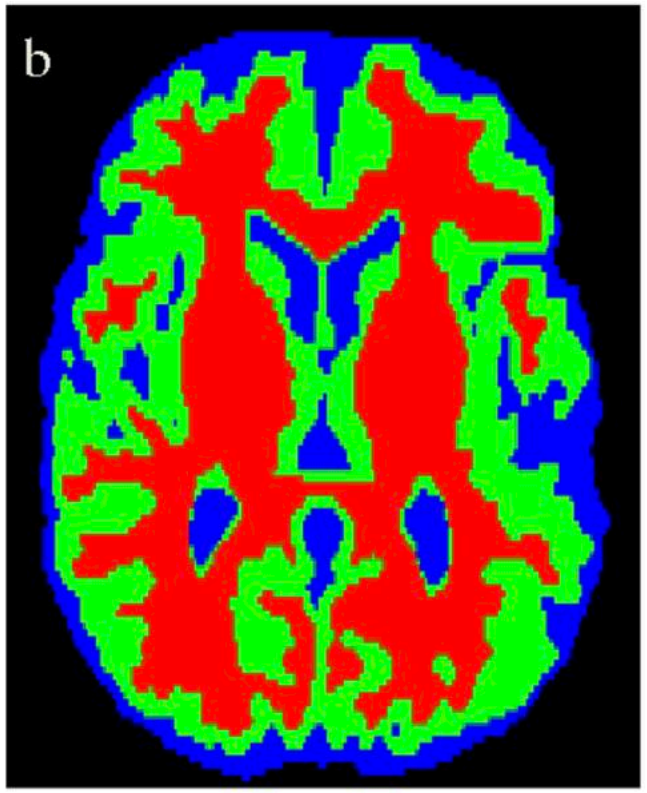
Image-brightness > -5

```
cv.adaptiveThreshold(img,255,cv.ADAPTIVE_THRESH_GAUSSIAN_C,
        cv.THRESH_BINARY,11,2)
```

Task 1: semantic segmentation

- Hard to manually engineer it → Machine learning (next week)

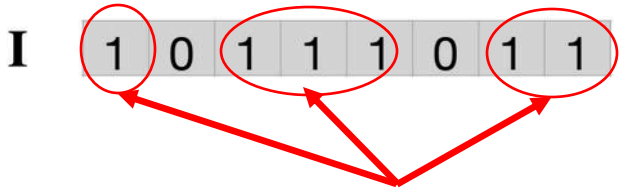




0: background
1: white matter
2: gray matter
3: fluid

Task 2: instance segmentation

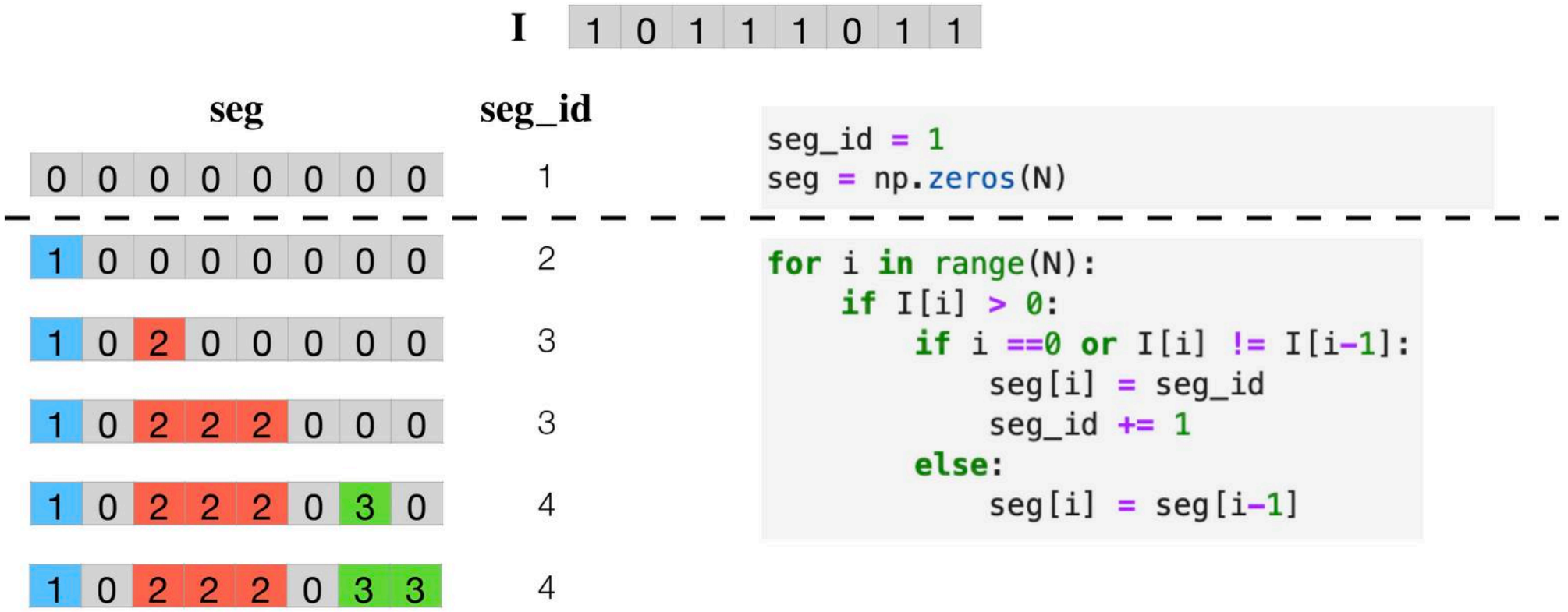
- Connected Component: 1D case
  - Goal: for the input binary image, assign pixels that are “connected” to the same label



A **Connected Component** is a **group of connected pixels** that share the same intensity or label.

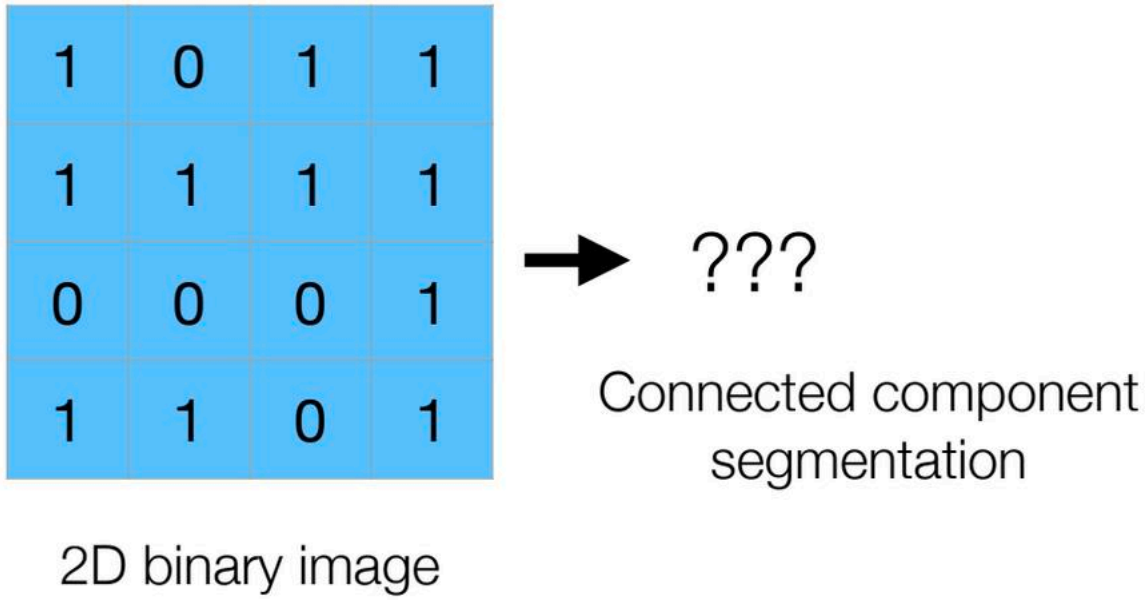
Task 2: instance segmentation

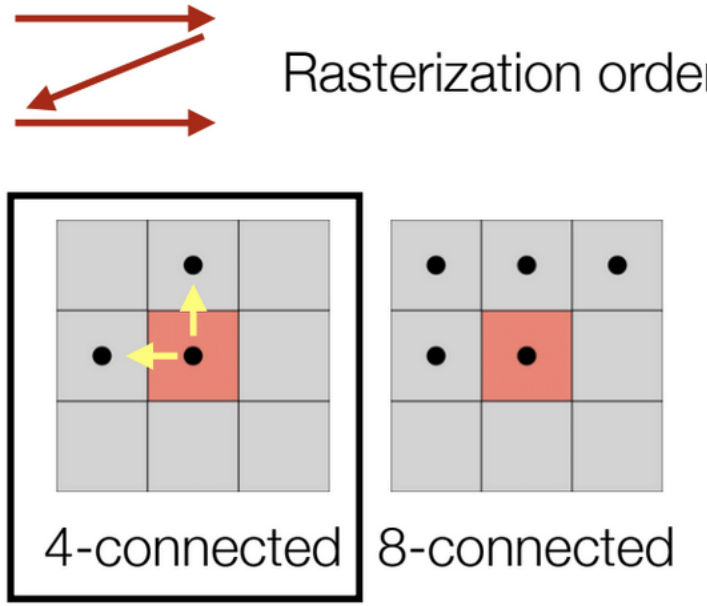
- Method 1: Connected Component: 1D case
  - Goal: for the input binary image, assign pixels that are “connected” to the same label



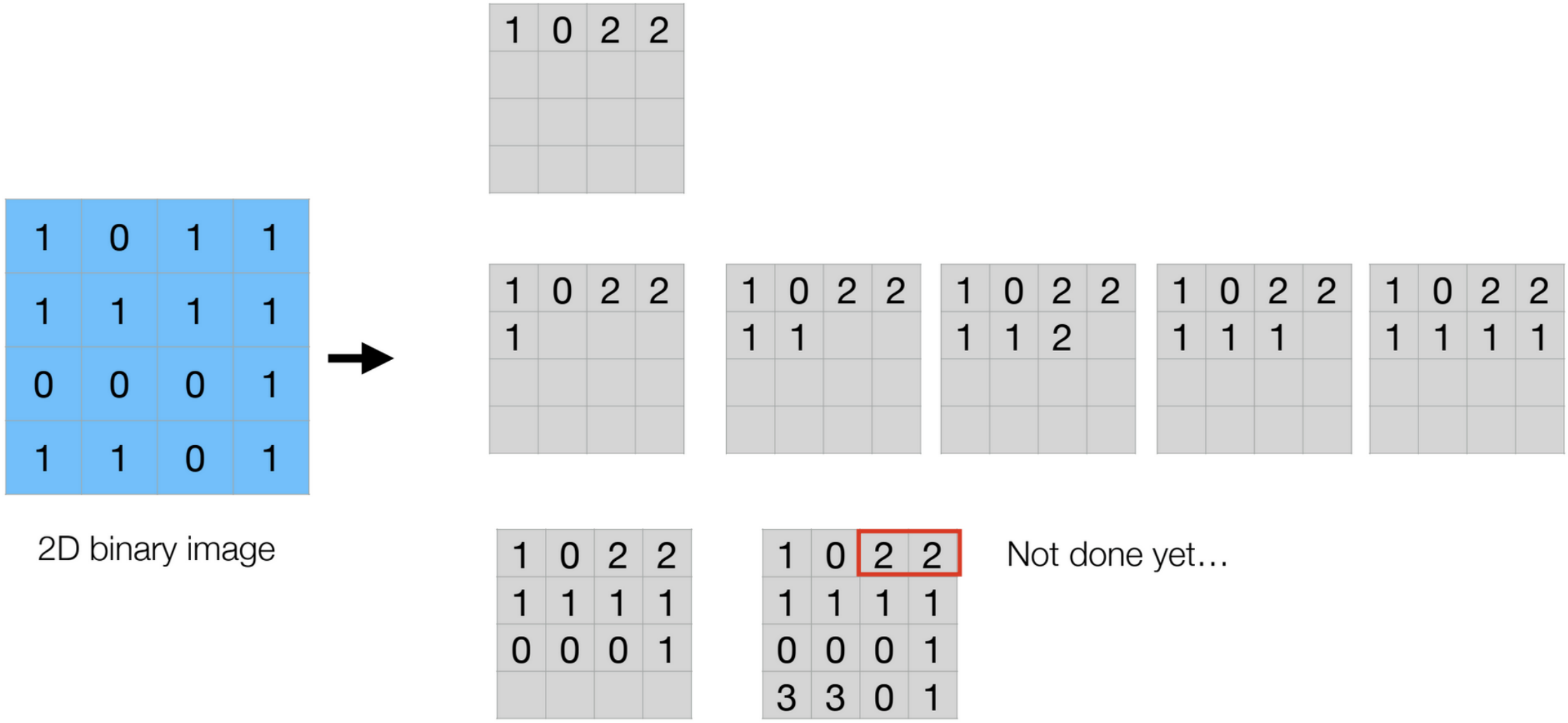
Task 2: instance segmentation

- Run through the algorithm for a 2D case
- Intuition: greedy approach
  - Assign the label for each pixel in the scan line order
  - For each non-zero pixel, check its left and then up neighbor's pixel value





Task 2: instance segmentation



Task 2: instance segmentation

```
from scipy.ndimage import label
from skimage.measure import label
seg_inst = label(seg_semantic)
```

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

2D binary image

Many algorithms to fix it

- Raster scan + labels merge (union find)
- Flood-fill
  - Breadth-first
  - Depth first
  - Line based

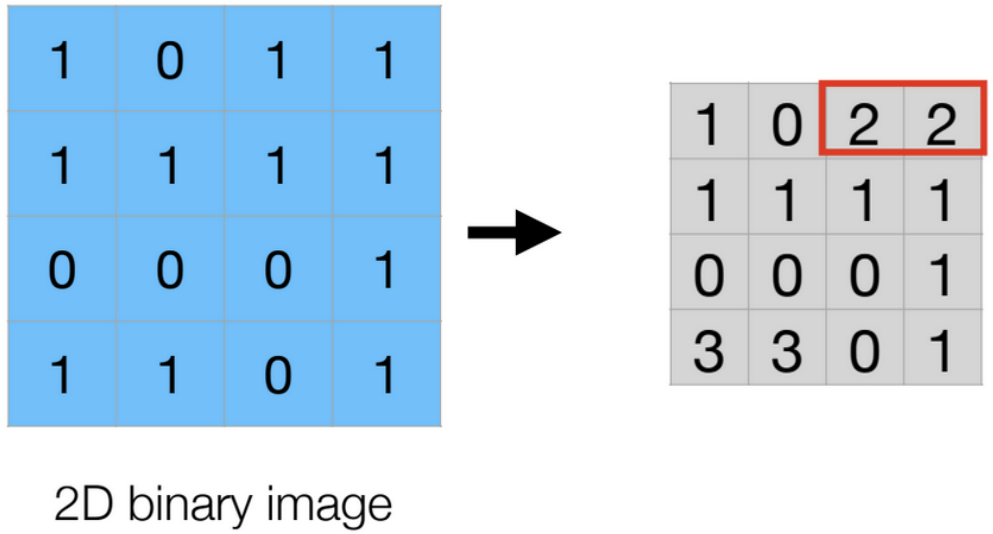
1	0	2	2
1	1	1	1
0	0	0	1

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

Not done yet…

Task 2: instance segmentation

```
from skimage.measure import label
seg_inst = label(seg_semantic)
```

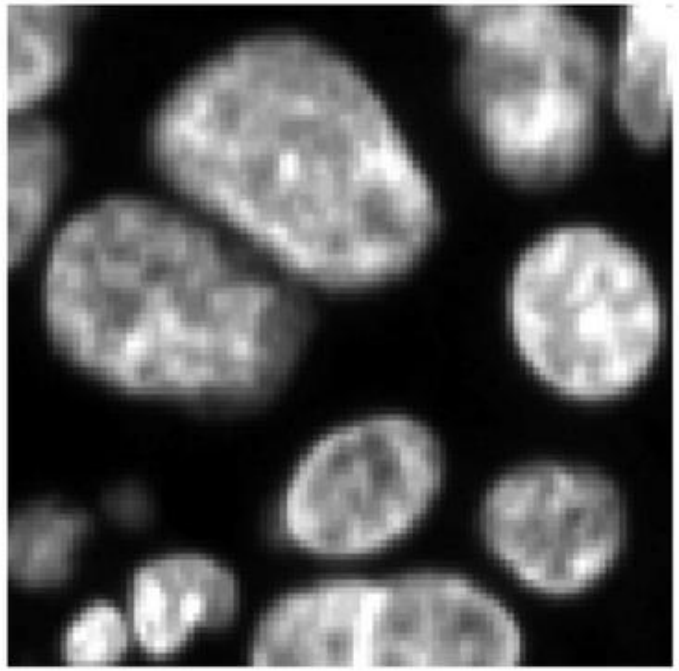


2D binary image

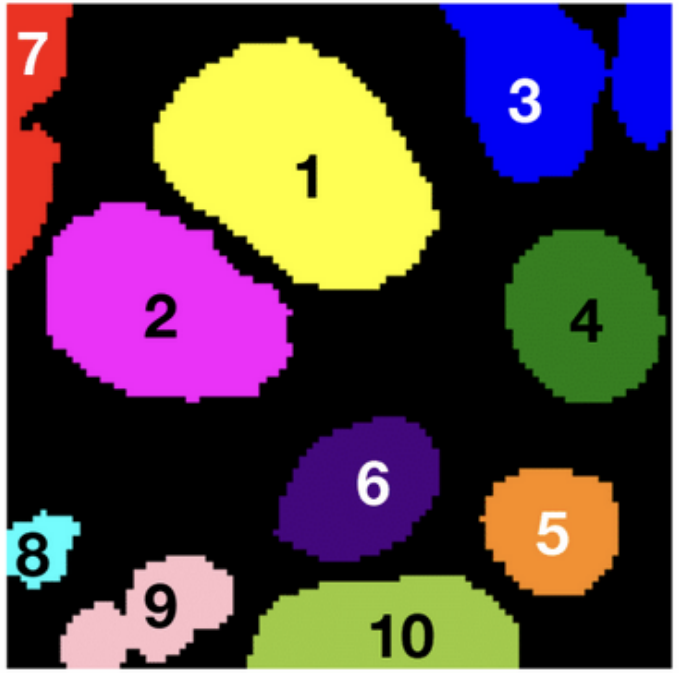
```
for r in range(N):
    for c in range(N):
        if I[r,c] > 0:
            if I[r,c]!=I[r-1,c] and I[r,c]!=I[r,c-1]:
                seg[r,c] = seg_id
                seg_id += 1
            elif I[r,c]==I[r-1,c]: # assign to up
                seg[r,c] = seg[r-1,c]
            elif I[r,c]==I[r,c-1]: # assign to left
                seg[r,c] = seg[r,c-1]
```

Task 2: instance segmentation

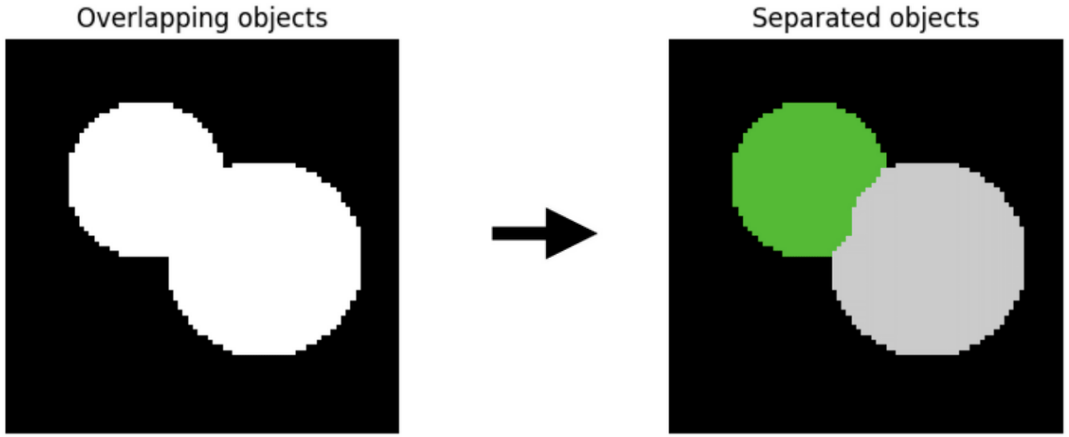
- Common problem: False merge error



Image



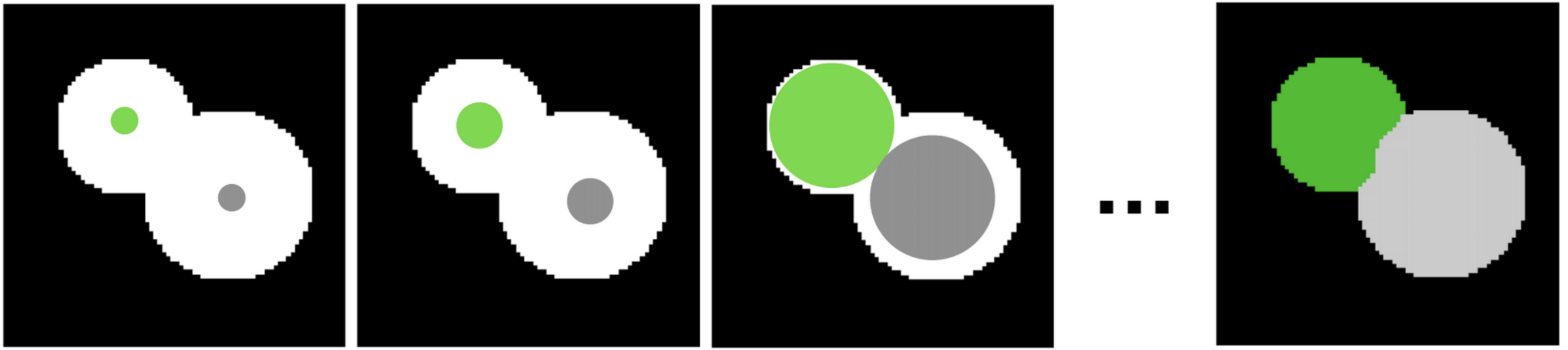
Instance segmentation



How to separate false merges?

Task 2: instance segmentation

- Common problem: False merge error
- Idea: known object center + flood filling



Known object centers

Flood it once

Stop in the direction until touches

Until all pixels are flooded

Task 2: instance segmentation

- Method 2: Watershed
  - Distance transform

```
from scipy import ndimage as ndi
distance = ndi.distance_transform_edt(image)
```

Binary image

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

Matrix representation

0	0	0	0	0	0	0	0
0	1	1	0	0	0	0	0
0	1	1	1	1	1	1	0
0	0	0	1	2	2	1	0
0	0	0	1	2	2	1	0
0	0	0	1	1	1	1	0
0	0	0	0	0	0	0	0

Distance to the boundary

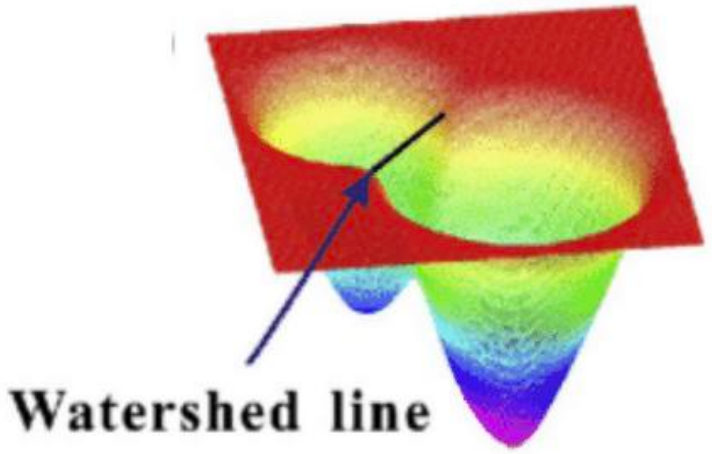
Task 2: instance segmentation

- Method 2: Watershed
  - Distance transform

Step 1: Distance transform

```
from scipy import ndimage as ndi
distance = ndi.distance_transform_edt(image)
```

Binary image



Distance transform (3D view)

Task 2: instance segmentation

- Method 2: Watershed
  - Distance transform
  - Find local optima

Step 2: Find local optima

```
from skimage.feature import peak_local_max
coords = peak_local_max(distance,\
                        footprint=np.ones((5, 5)),\
                        labels=image)
mask = np.zeros(distance.shape, dtype=bool)
mask[tuple(coords.T)] = True
markers, _ = ndi.label(mask)
```

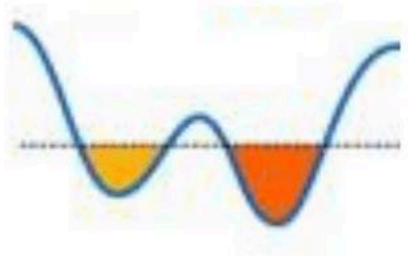
a) Optima's pixel location (within a [5,5] window)

b) Make a binary image with optima

c) Each optima has its own seg id

Binary image

Distance transform (3D view)



Start from local optima

Until first touch

Watershed segmentation (1D view)

Task 2: instance segmentation

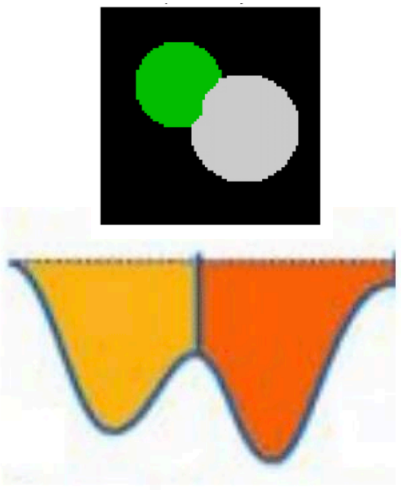
- Method 2: Watershed
  - Distance transform
  - Find local optima
  - Watershed transform

Step 2: Watershed transform

```
from skimage.segmentation import watershed
labels = watershed(-distance, markers, mask=image)
```

Binary image

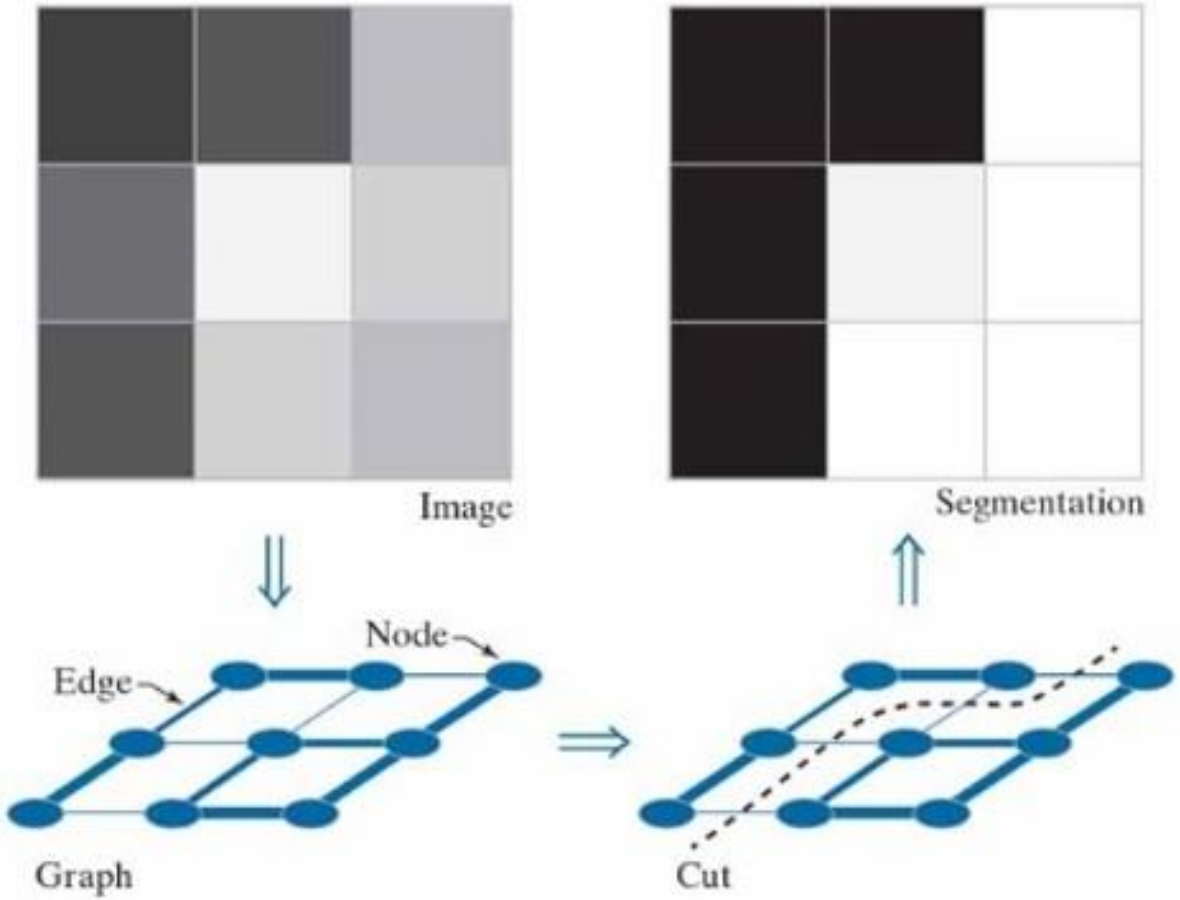
Distance transform (3D view)



Watershed segmentation (1D view)

Task 2: instance segmentation

- Method 3: Graph-based method
  - Image as graph $G = (V, E)$, $E \subseteq V \times V$, W is a weight matrix associated with E



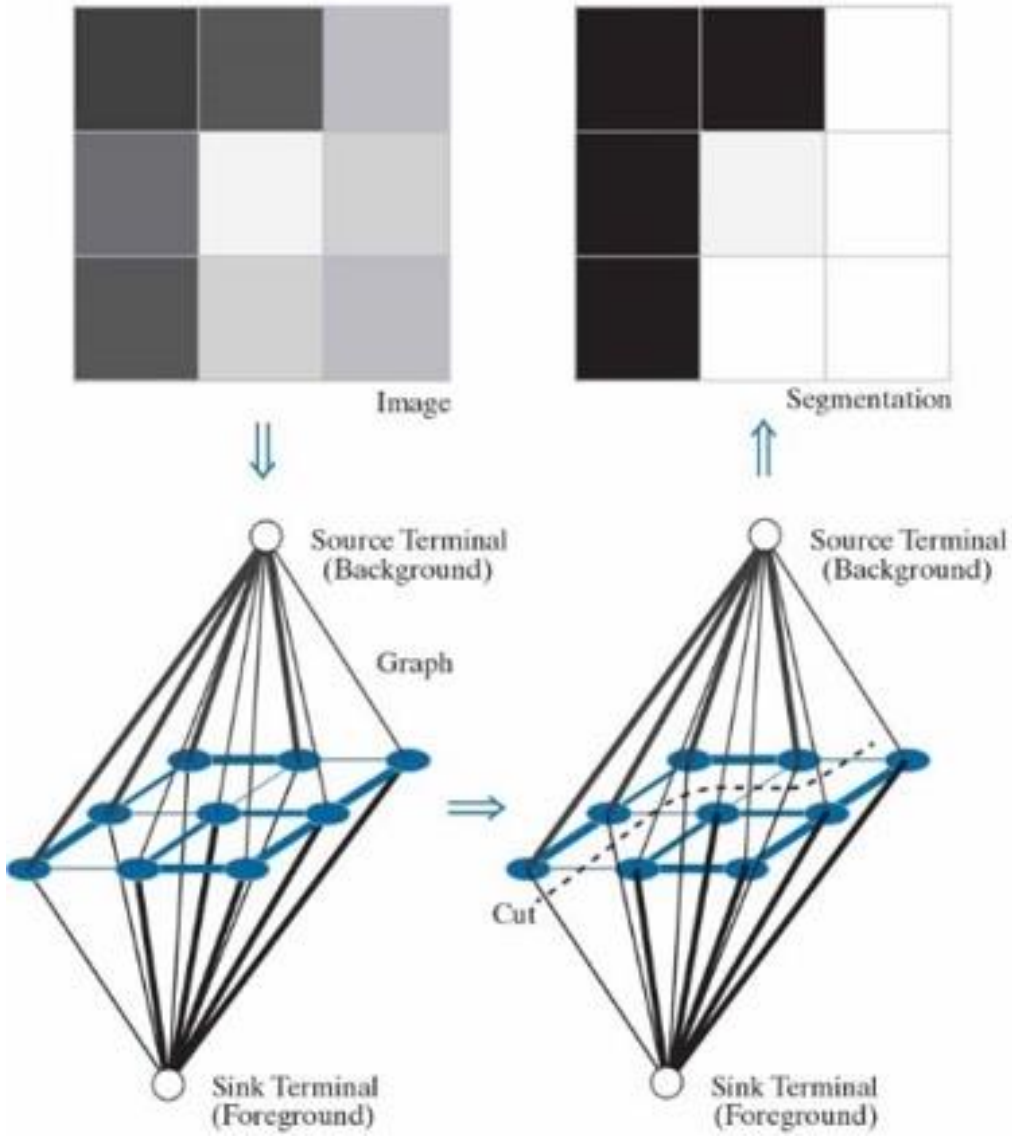
- ❑ The nodes V
  - correspond to the pixels in the image
- ❑ Edges E
  - Usually between adjacent pixels using 4-connectivity
  - + diagonal edges
  - + more general edges: between every pair of pixels
- ❑ Weight E
  - Example:

$$w(i,j) = \frac{1}{\left|I(n_i) - I(n_j)\right| + c}$$

  - Cutting the graph along its weak edges

Graph algorithm 1: min cut / max flow

- Source & sink terminal nodes
  - Not part of the image
  - Role: to associate with each pixel a probability that it is a background or foreground pixel
- t-links
  - Connected to all nodes in the graph via unidirectional links
  - The weights of the t-links is the associated foreground/background probability

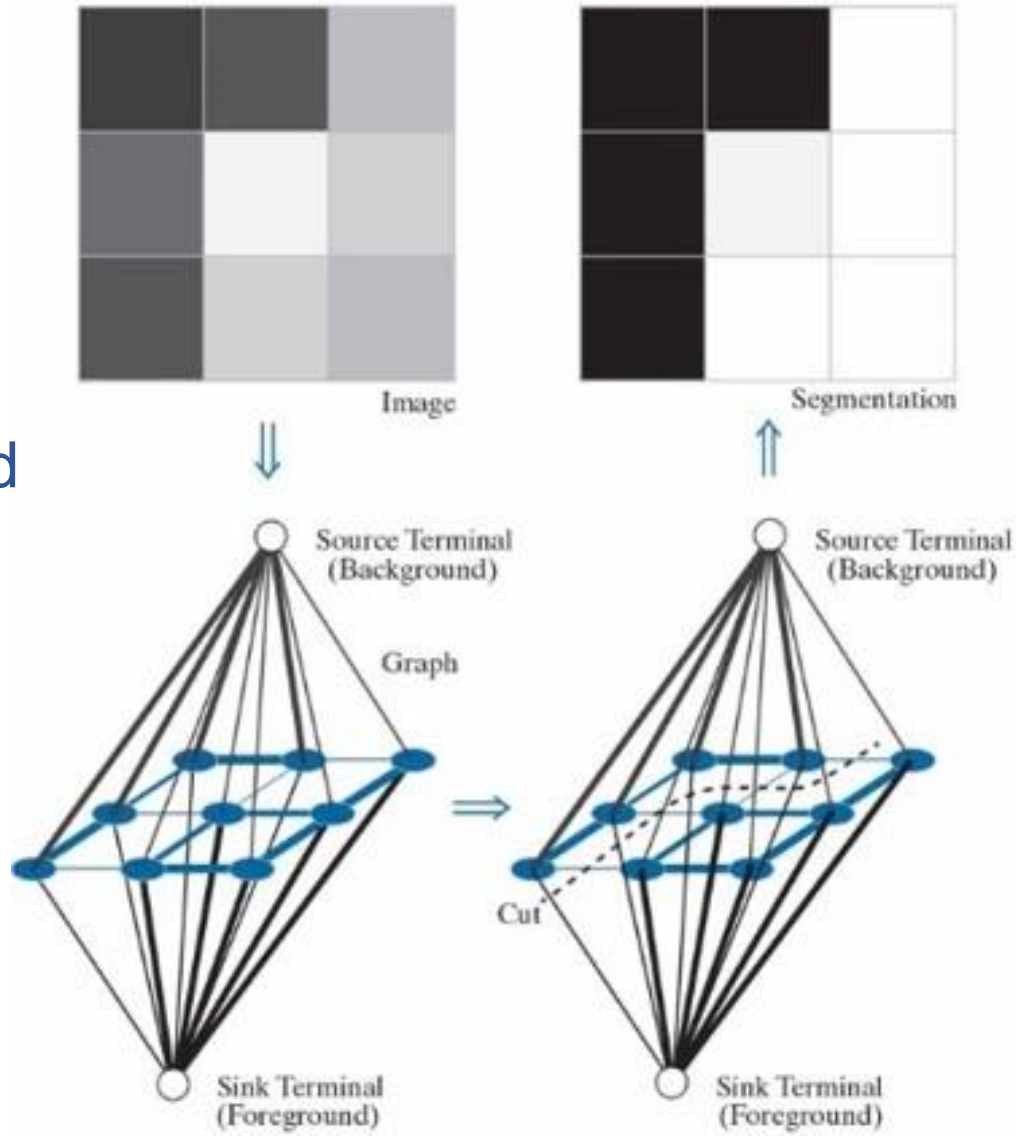


Graph algorithm 1: min cut / max flow

- Problem formulation
  - In a flow network, the maximum amount of flow passing from the *source* to the *sink* is equal to the minimum cut
  - This minimum cut is defined as the smallest total weight of the edges that, if removed, would disconnect the sink from the source

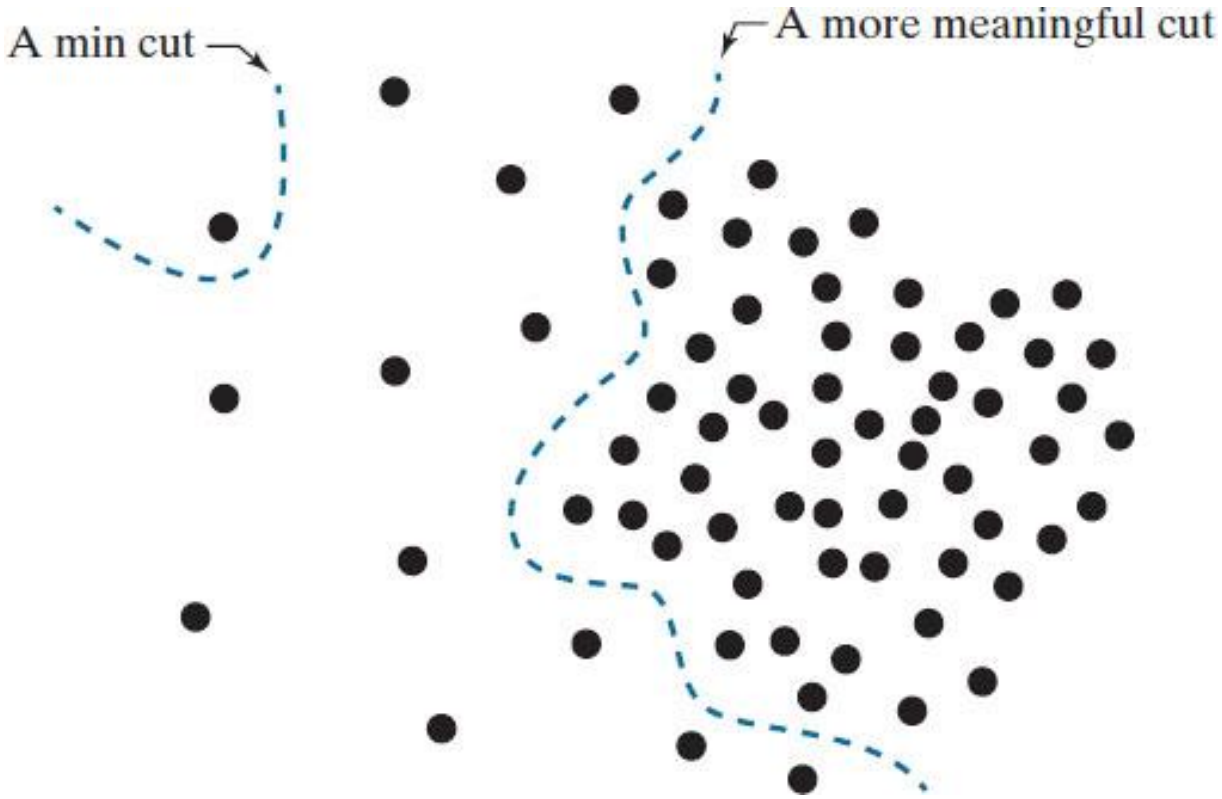
$$cut(A, B) = \sum_{u \in A,\, v \in B} w(u, v)$$

$$A \cup B = V \text{ and } A \cap B = \emptyset$$



Problem with min cuts

- Min. cuts favor cutting small sets of isolated nodes in a graph



In this example, the similarity between pixels is defined as their spatial proximity, which results in two distinct regions.

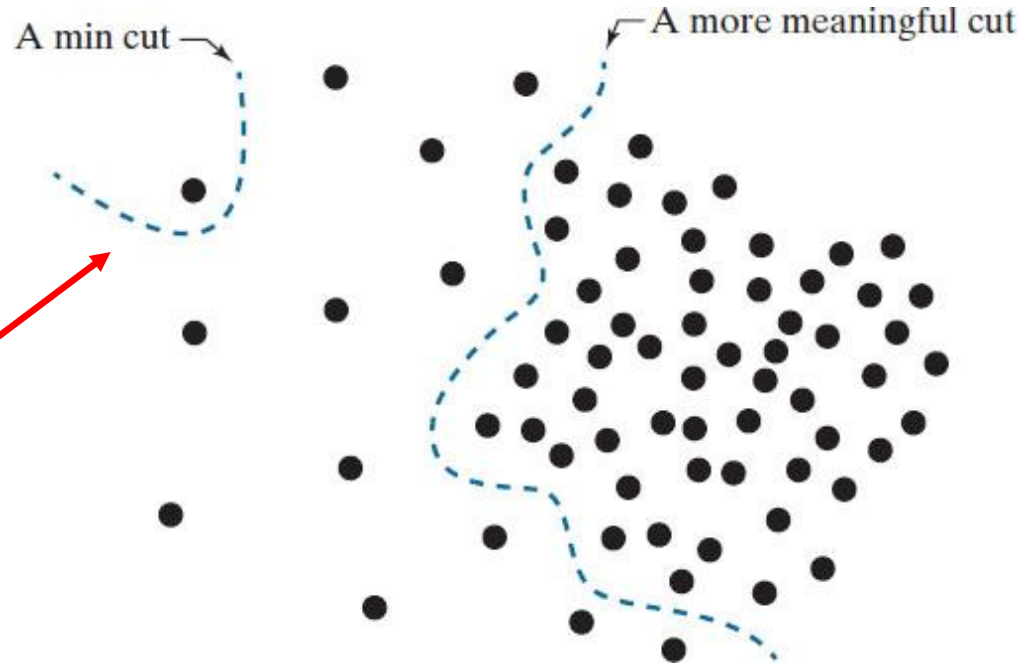
Any cut that partitions out individual points on the left of the figure will have a smaller cut value than a cut that properly partitions the points into two groups based on their proximity.

Graph algorithm 2: normalized min-cut

- $Ncut(A,B) = \frac{cut(A,B)}{assoc(A,V)} + \frac{cut(A,B)}{assoc(B,V)}$
- $assoc(A,V) = \sum_{u \in A,\, z \in V} w(u,z)$
  - sum of the weights of all the edges from the nodes of subgraph A to the nodes of the entire graph
  - $assoc(A,V)$ is simply the cut of A from the rest of the graph
- By using $Ncut(A,B)$ instead of $cut(A,B)$, the cut that partitions isolated points will no longer have small values

$$assoc(A,V) = cut(A,B)$$

$$Ncut(A,B) \geq 1$$

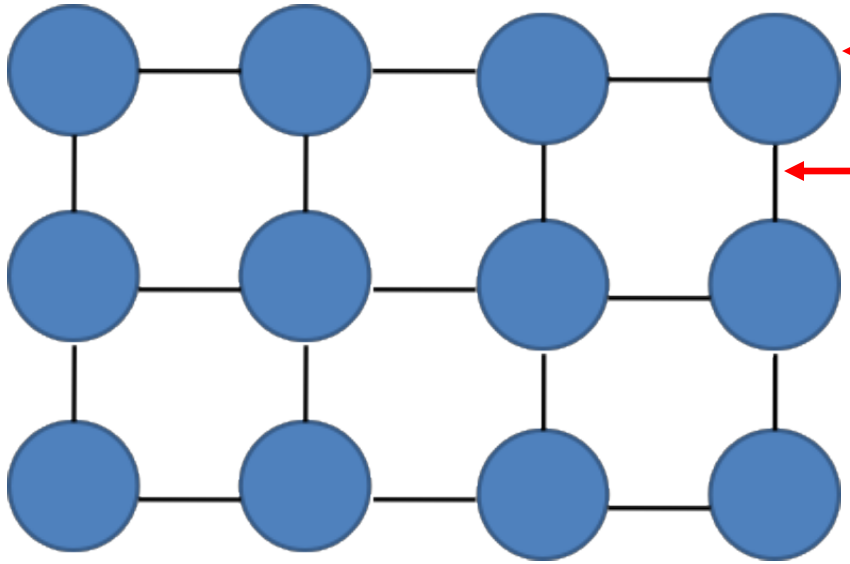


Graph algorithm 2: normalized min-cut

- Pro
  - Flexible to choice of affinity matrix
  - Generally, works better than other methods we've seen so far
- Con
  - Can be expensive, especially with many cuts
  - Bias toward balanced partitions
  - Constrained by affinity matrix model

Graph algorithm 3: Markov random field

- Loss = Data term (node) + Smoothness term (edge)



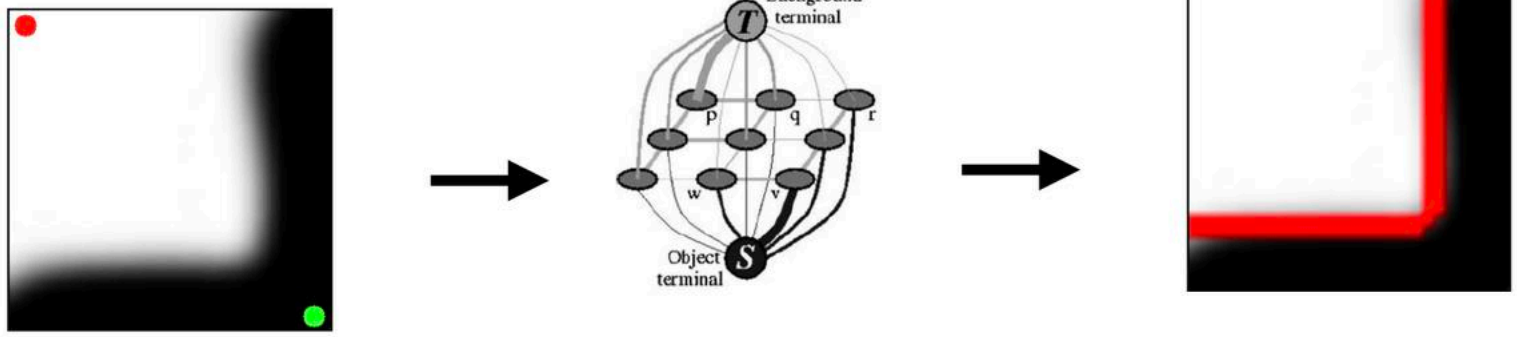
Node y_i: pixel label

Edge: constrained pairs

$$Energy(\mathbf{y};\theta,data) = \sum_i \psi_1(y_i;\theta,data) \sum_{i,j\in edges} \psi_2(y_i,y_j;\theta,data)$$

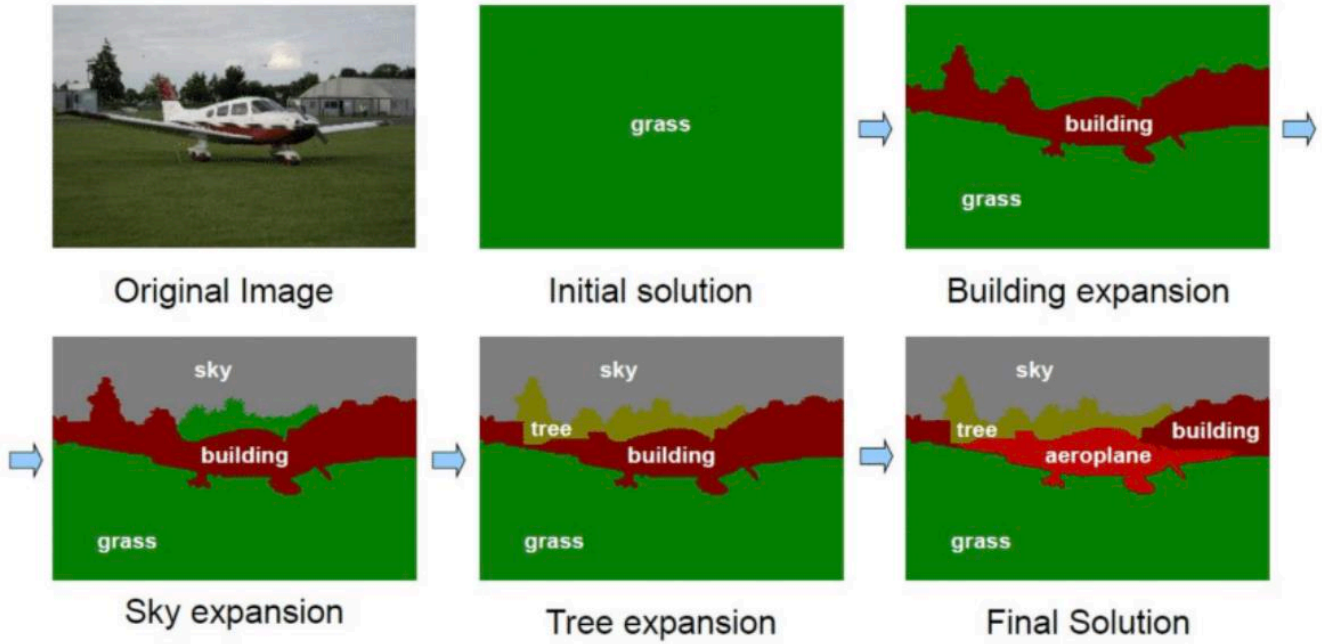
Cost to assign a label to each pixel

Cost to assign a pair of labels to connected pixels



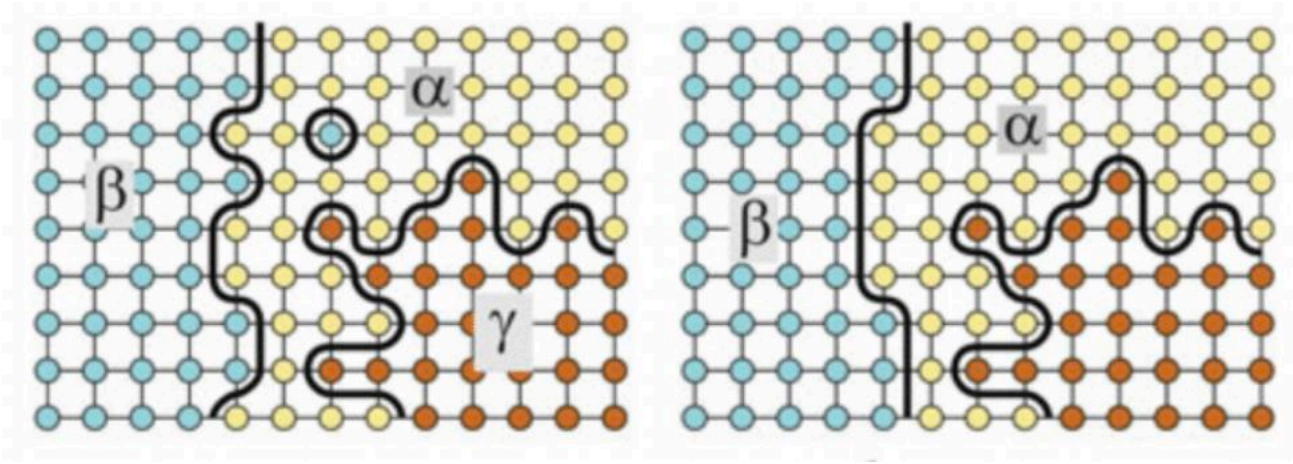
Optimization: graph cut (many solvers)

- One popular greedy method: Start from an initial segmentation, randomly select 1 or 2 segments to optimize
- α-expansion (1 seg at a time)
  - Given a label α, a move from a labeling L_1 to a labeling L_2 is called an α-expansion move if the only difference between L_1 and L_2 is that
    - some vertices that were not labeled α in L_1 are labeled α in L_2



Optimization: graph cut (many solvers)

- One popular greedy method: Start from an initial segmentation, randomly select 1 or 2 segments to optimize
- $\alpha - \beta$ swap (2 segs at a time)
  - Given a pair of labels α, β, a move from a labeling L_1 to a labeling L_2 is called an $\alpha - \beta$ swap move if the only difference between L_1 and L_2 is that
    - some vertices that were not labeled α in L_1 are labeled β in L_2, and
    - some vertices that were not labeled β in L_1 are labeled α in L_2.



GrabCut algorithm: Segmentation from N scribbles

input image

input mask

- bounding box initialization: cv2.GC_INIT_WITH_RECT
- mask initialization: cv2.GC_INIT_WITH_MASK

Temporary array used by GrabCut internally when modeling the background

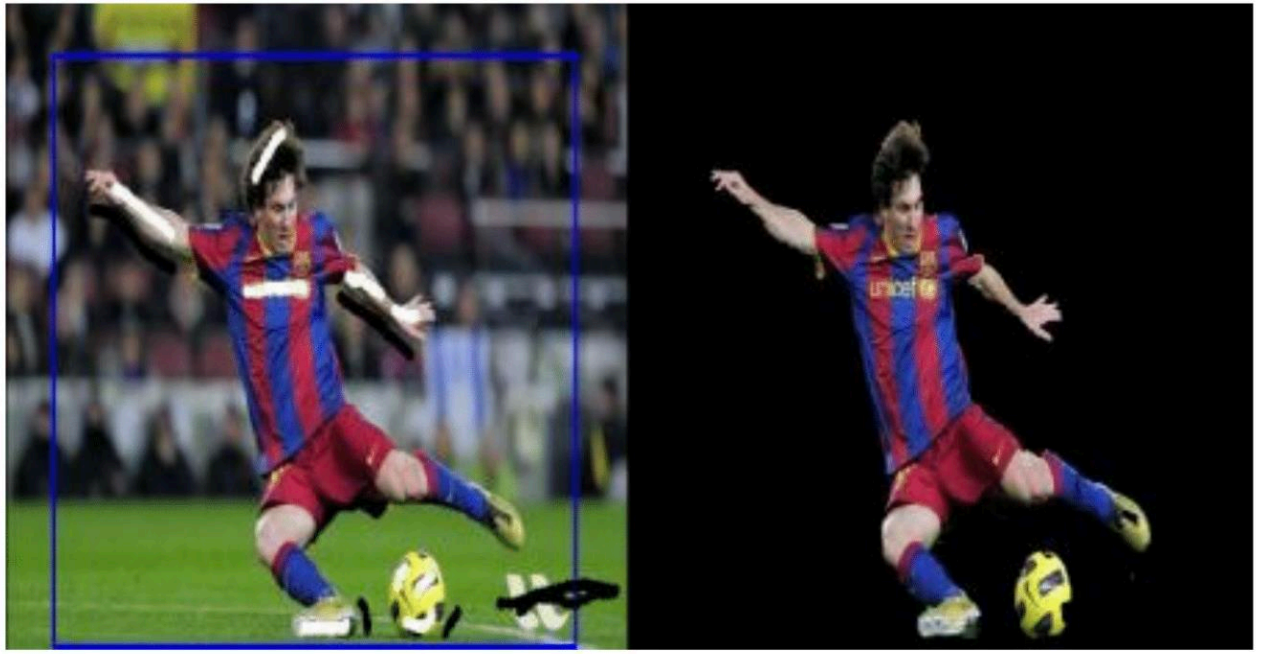
Number of iterations

cv2.grabCut(img, mask, rect, bgdModel, fgdModel, iterCount[, mode])

→ mask, bgdModel, fgdModel

```
cv.grabCut(img,mask,rect,bgdModel,fgdModel,5,cv.GC_INIT_WITH_RECT)
```

Temporary array used by GrabCut internally when modeling the foreground



MRV-based graph cut summary

- Pros
  - Very powerful, get global results by defining local interactions
  - Very general
  - Rather efficient
  - Becoming more or less standard for many segmentation problems (GrabCut was 2004!)
- Cons
  - Only works for sub modular energy functions (binary)
  - Only approximate algorithms work for multi-label case

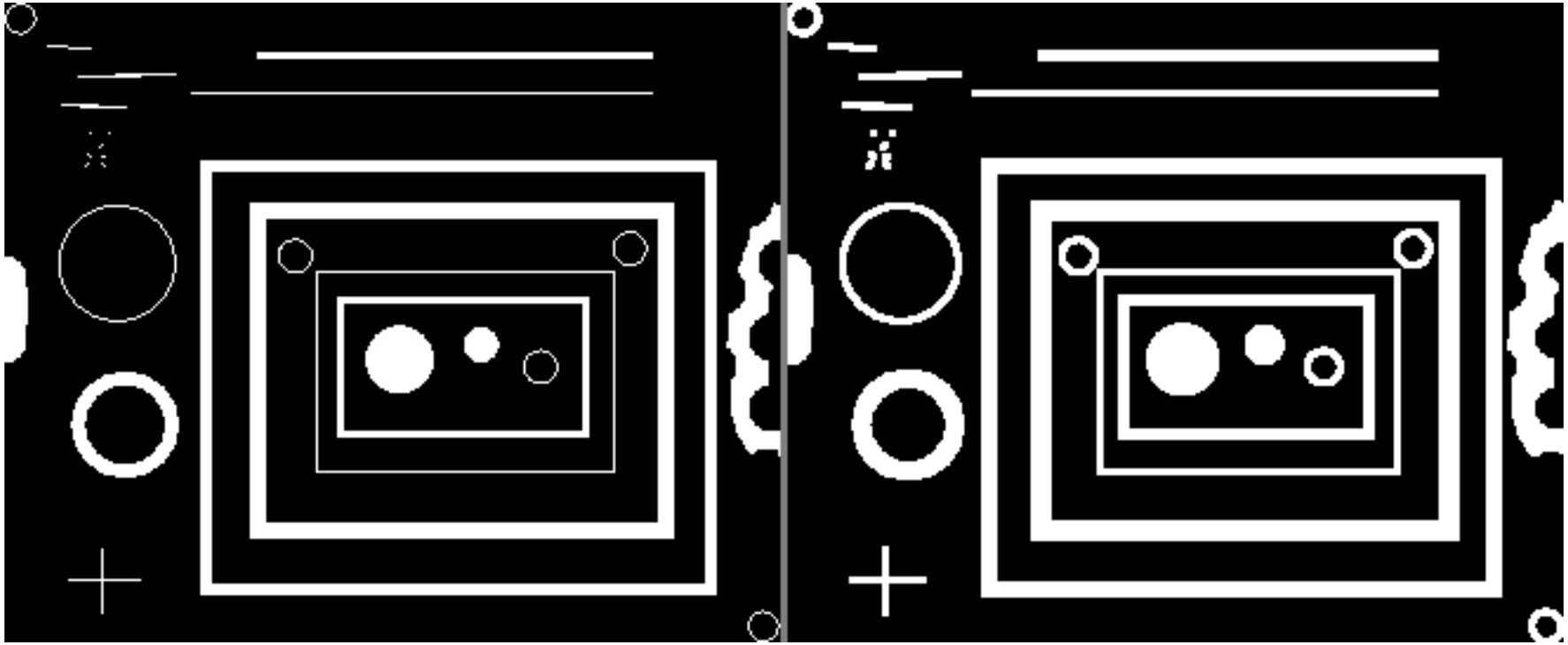
Post-processing

- Common morphological operations

- Two basic operations
  - Erosion
  - Dilation
- Other operations
  - Thinning/thickening
  - Hole filling

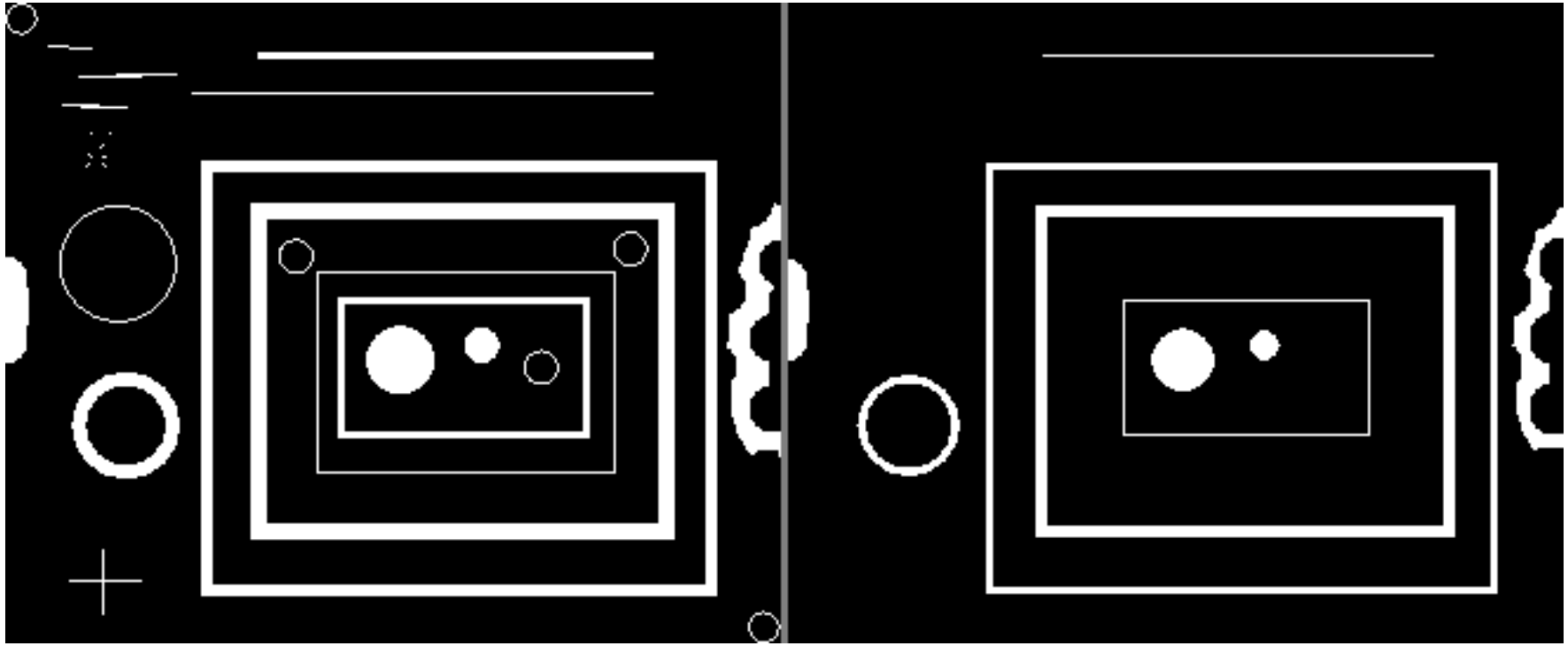
Dilation

- The value of the output pixel is the *maximum* value of all pixels in the neighborhood. In a *binary image*, a pixel is set to 1 if any of the neighboring pixels have the value 1.
- Morphological dilation makes objects more visible and fills in small holes in objects. Lines appear thicker, and filled shapes appear larger.



Erosion

- The value of the output pixel is the *minimum* value of all pixels in the neighborhood. In a *binary image*, a pixel is set to 0 if any of the neighboring pixels have the value 0.
- Morphological erosion removes floating pixels and thin lines so that only substantive objects remain. Remaining lines appear thinner and shapes appear smaller.

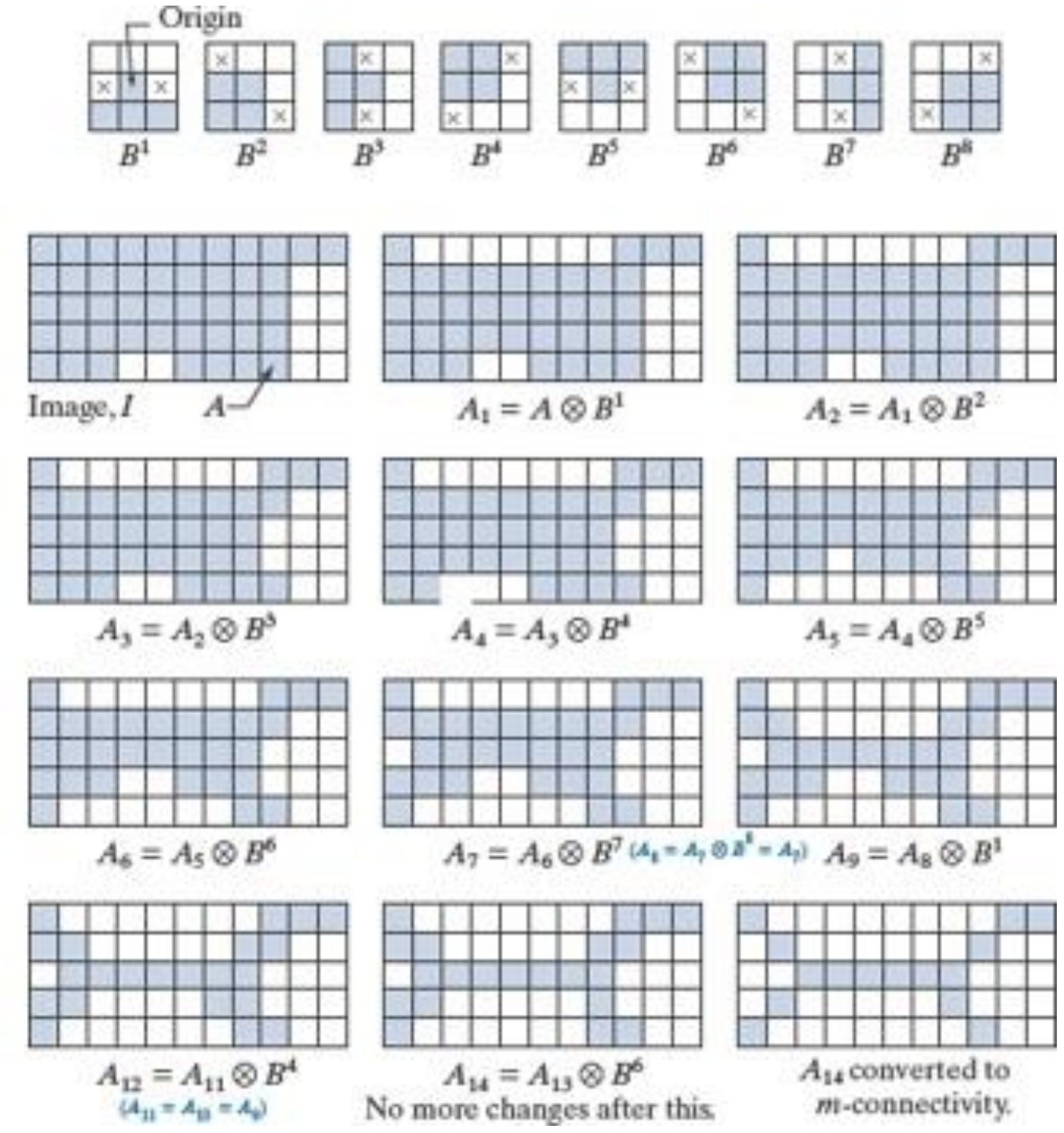


Thinning

$$A \otimes B = A - (A \circledast B)$$

$$B = \{B^1, B^2, B^3, \cdots, B^n\}$$

$$A \otimes B = \left(\left(\left((A \otimes B^1) \otimes B^2\right) \cdots\right) \otimes B^n\right)$$

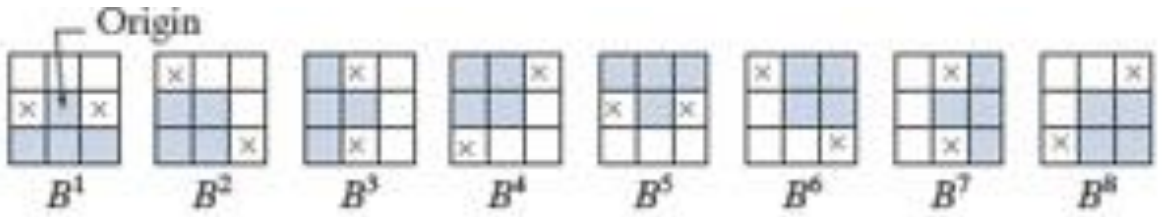


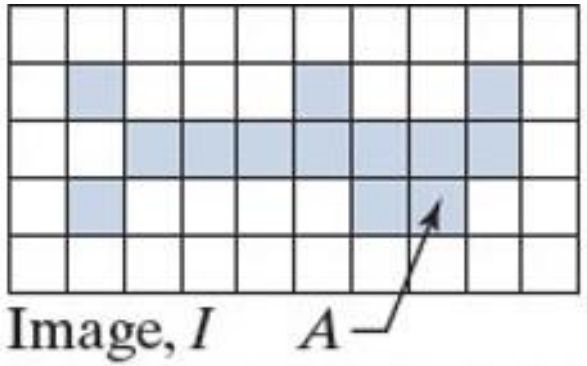
Thickening

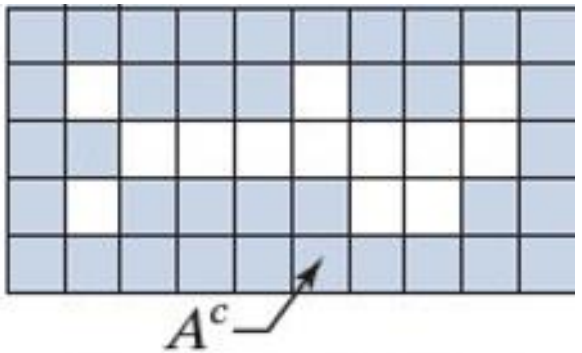
$$A \odot B = A \cup (A \circledast B)$$

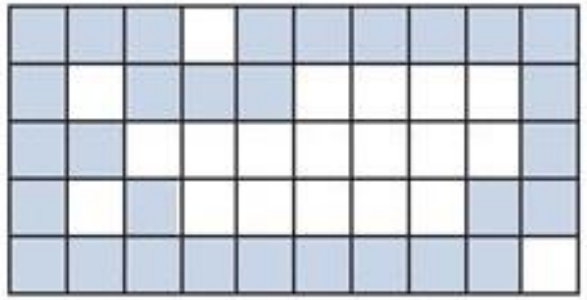
$$B = \{B^1, B^2, B^3, \cdots, B^n\}$$

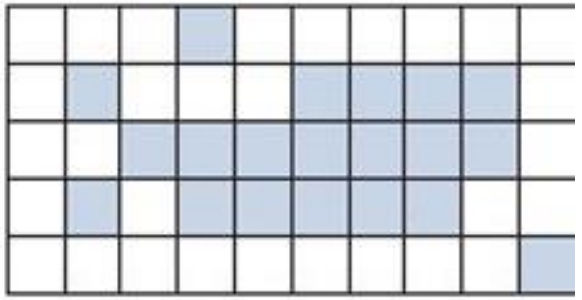
$$A \odot B = A^c \otimes B$$



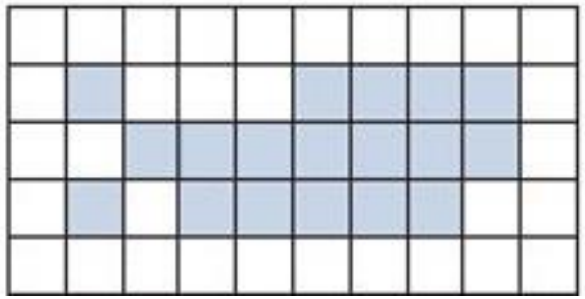








Final result, with no disconnected points



Application: boundary extraction

$$\beta(A) = A - (A \ominus B)$$

Application: hole filling

- Hole: a background region surrounded by a connected foreground pixels
- Objective: given a point in a hole, fill the hole with foreground pixels

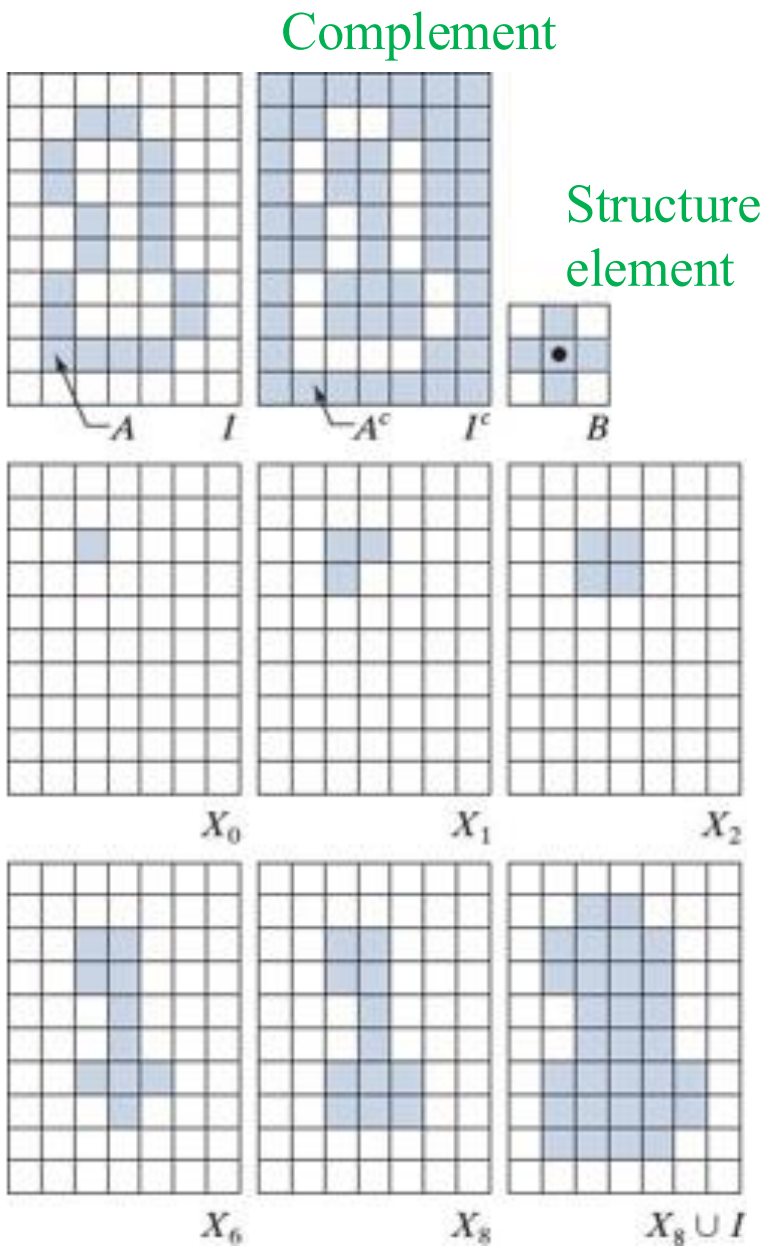
X_0 is a set of all 0s except the selected background point
A is the set of foreground boundary

Repeat

$$X_k = (X_{k-1} \oplus B) \cap A^c, k = 1,2,3, \cdots$$

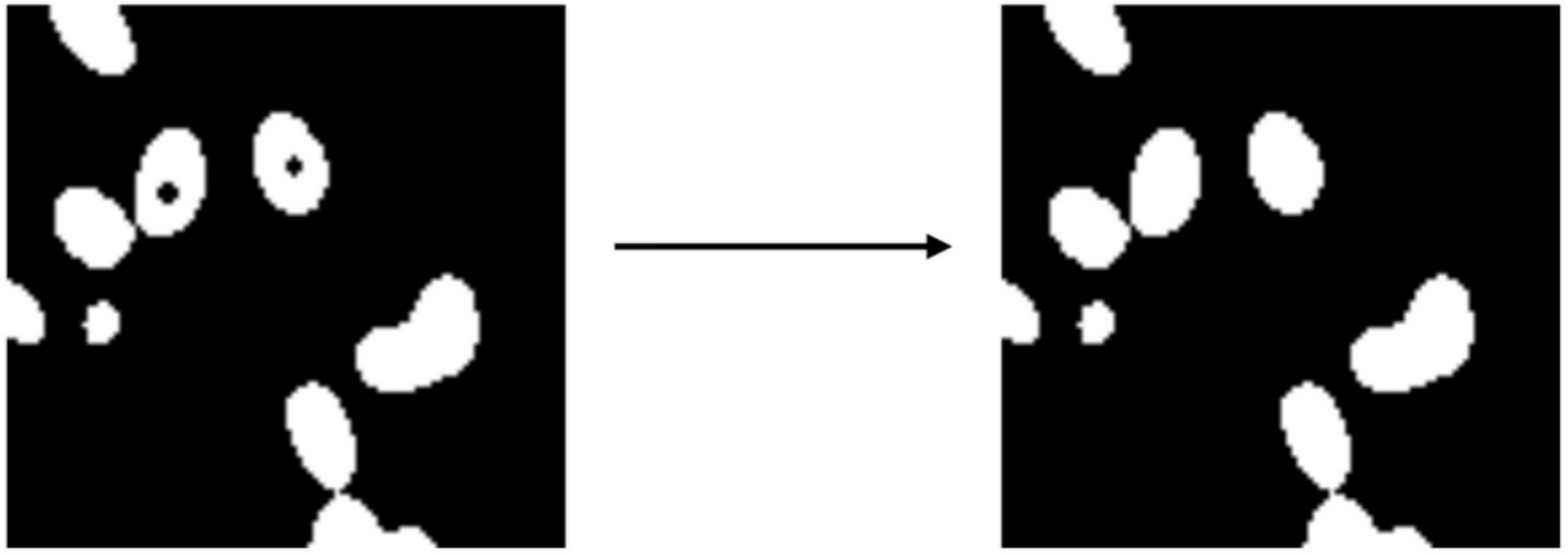
Until

$$X_k = X_{k-1}$$



Example

```
from scipy.ndimage.morphology import binary_fill_holes
im_full = binary_fill_holes(im)
```



Segmentation post-processing

Morphology

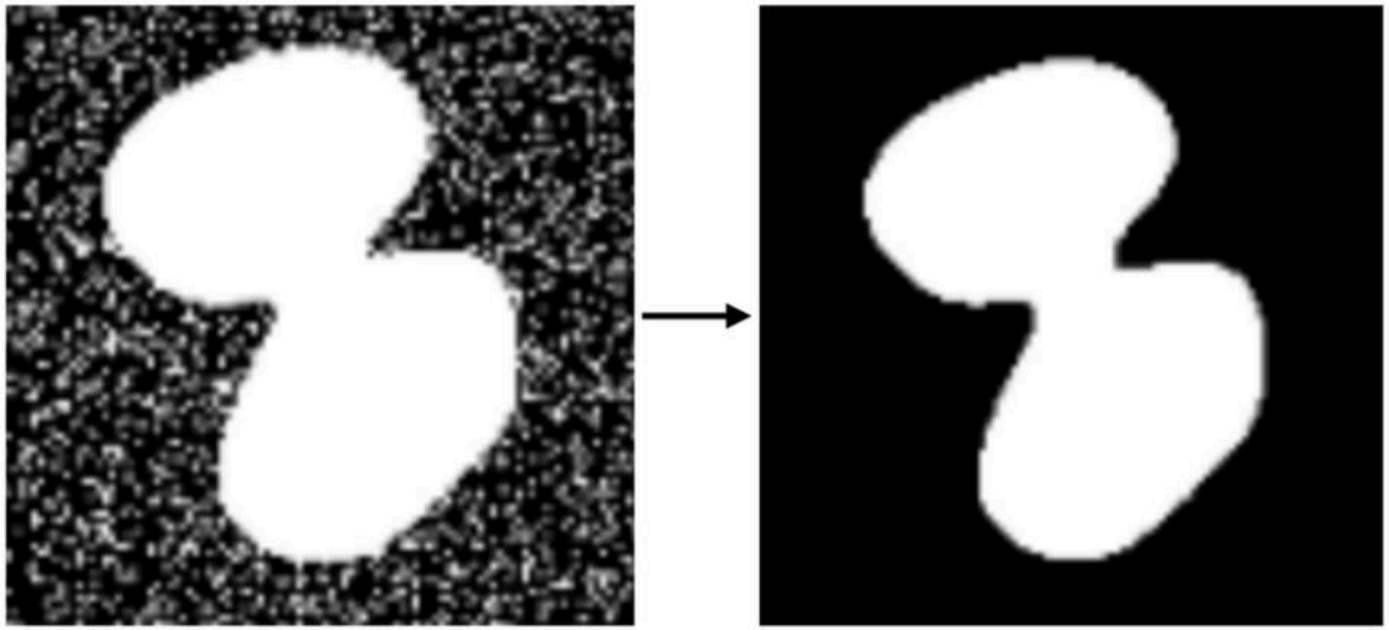
`binary_closing` (input[, structure, ...])	Multidimensional binary closing with the given structuring element.
`binary_dilation` (input[, structure, ...])	Multidimensional binary dilation with the given structuring element.
`binary_erosion` (input[, structure, ...])	Multidimensional binary erosion with a given structuring element.
`binary_fill_holes` (input[, structure, ...])	Fill the holes in binary objects.
`binary_hit_or_miss` (input[, structure1, ...])	Multidimensional binary hit-or-miss transform.
`binary_opening` (input[, structure, ...])	Multidimensional binary opening with the given structuring element.
`binary_propagation` (input[, structure, mask, ...])	Multidimensional binary propagation with the given structuring element.
`black_tophat` (input[, size, footprint, ...])	Multidimensional black tophat filter.
`distance_transform_bf` (input[, metric, ...])	Distance transform function by a brute force algorithm.
`distance_transform_cdt` (input[, metric, ...])	Distance transform for chamfer type of transforms.
`distance_transform_edt` (input[, sampling, ...])	Exact Euclidean distance transform.
`generate_binary_structure` (rank, connectivity)	Generate a binary structure for binary morphological operations.
`grey_closing` (input[, size, footprint, ...])	Multidimensional grayscale closing.
`grey_dilation` (input[, size, footprint, ...])	Calculate a greyscale dilation, using either a structuring element, or a footprint corresponding to a flat structuring element.
`grey_erosion` (input[, size, footprint, ...])	Calculate a greyscale erosion, using either a structuring element, or a footprint corresponding to a flat structuring element.
`grey_opening` (input[, size, footprint, ...])	Multidimensional grayscale opening.
`iterate_structure` (structure, iterations[, ...])	Iterate a structure by dilating it with itself.

Remove small noise

```
from scipy.ndimage.morphology import binary_erosion
im_denoise = binary_erosion(im, np.ones([3,3]), 1)
```

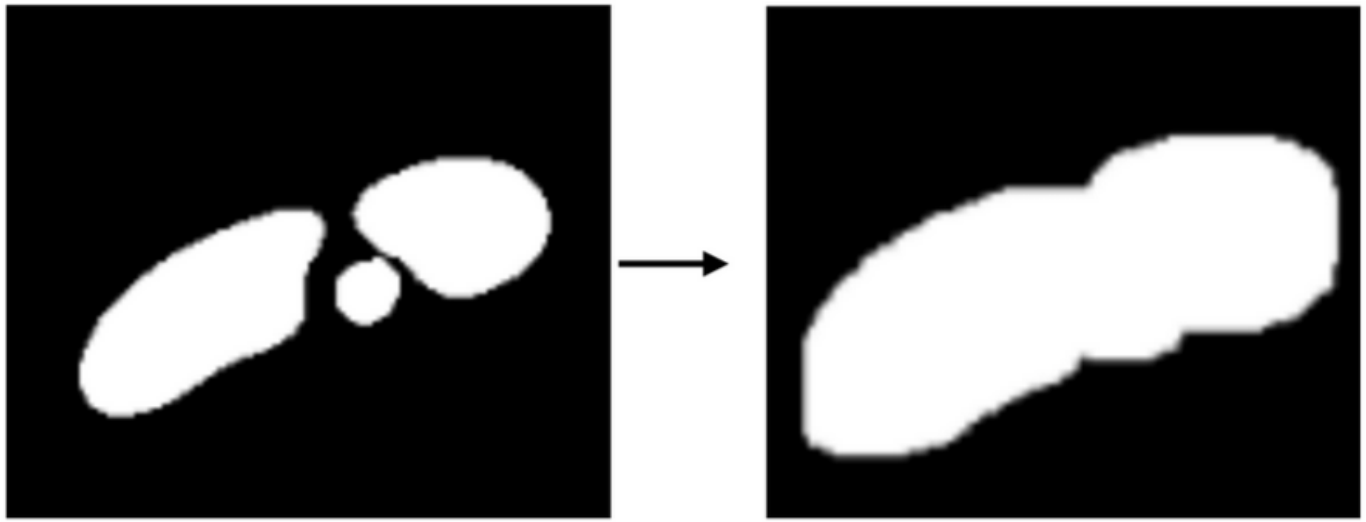
Erosion

- when the intensity of the center pixel is 1
- all the connected pixels are converted to 0



Merge false splits

```
from scipy.ndimage.morphology import binary_dilation
im_enlarge = binary_dilation(im, np.ones([3,3]), 5)
```



Dilation

- when the intensity of the center pixel is 1
- all the connected pixels are converted to 1

binary_closing(I)=binary_erosion(binary_dilation(I))

```
im_close = binary_closing(im, np.ones([3,3]), 5)
```

```
im_enlarge = binary_dilation(im, np.ones([3,3]), 5)
```

```
im_close = binary_erosion(im_enlarge, np.ones([3,3]), 5)
```

binary_opening(I)=binary_dilation(binary_erosion(I))

```
im_open = binary_opening(im, np.ones([3,3]), 5)
```

```
im_erosion = binary_erosion(im, np.ones([3,3]), 5)
```

```
im2 = binary_dilation(im_erosion, np.ones([3,3]), 5)
```

Caution: Only fix false merges with thin connections