Computer Graphics (CS/ECE 545)
Lecture 7: Morphology (Part 2) & Regions in Binary Images (Part 1)

Prof Emmanuel Agu

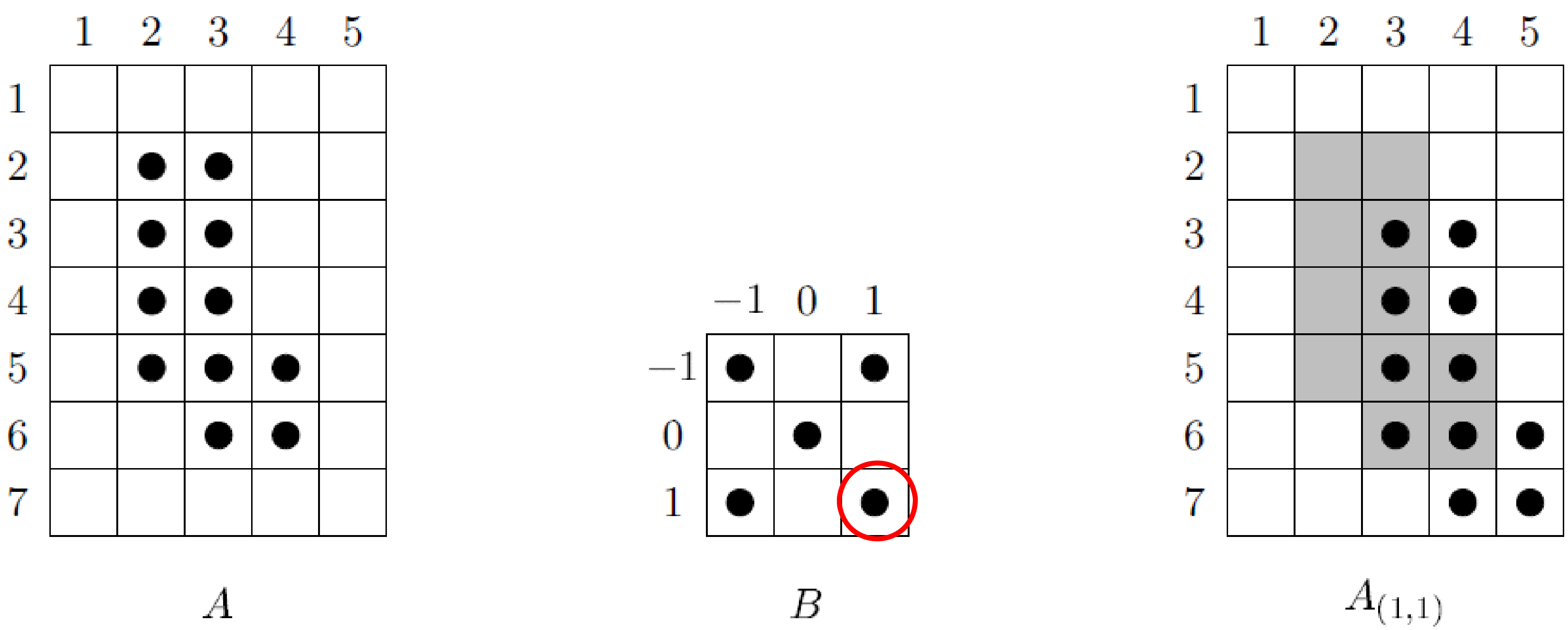
Computer Science Dept.

Worcester Polytechnic Institute (WPI)

Recall: Dilation Example

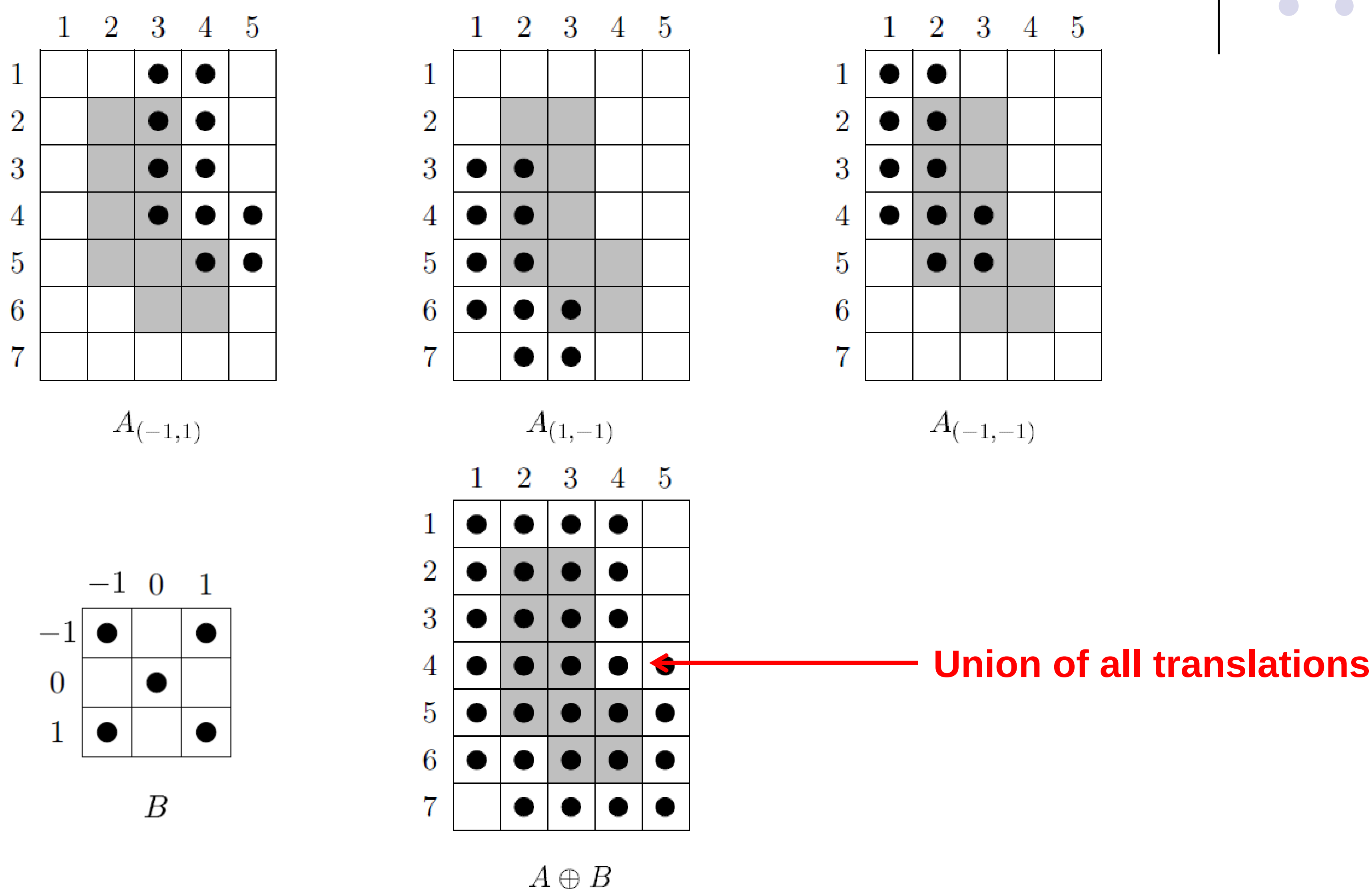
- For *A* and *B* shown below

$$B = \{(0,0), (1,1), (-1,1), (1,-1), (-1,-1)\}$$



Translation of *A* by (1,1)

Recall: Dilation Example

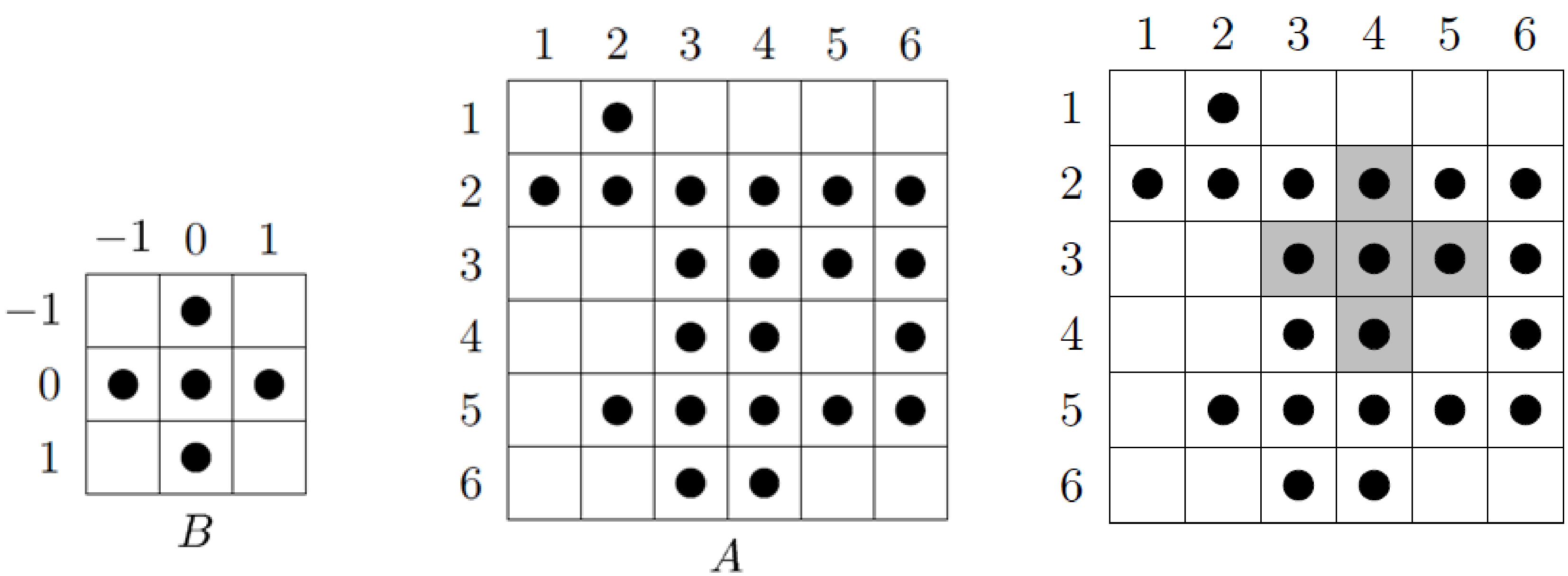


Recall: Erosion

- Given sets *A* and *B*, the **erosion of *A* by *B***

$$A \ominus B = \{w : B_w \subseteq A\}.$$

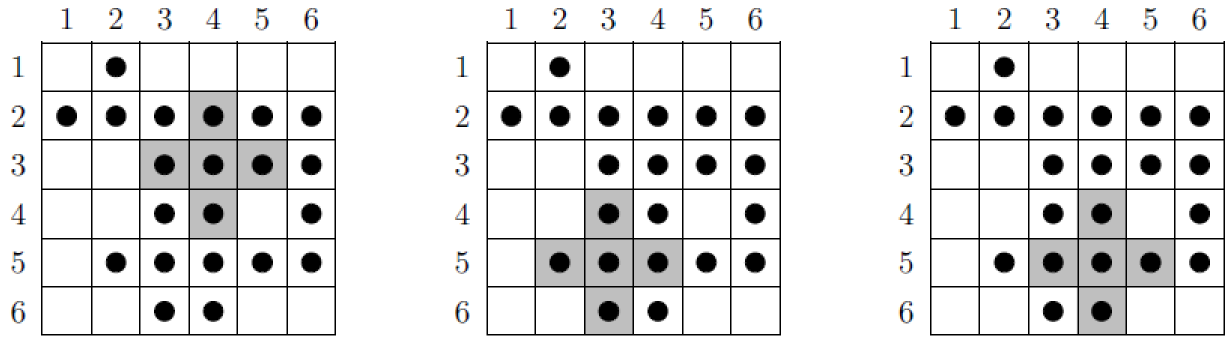
- Find all occurrences of *B* in *A*



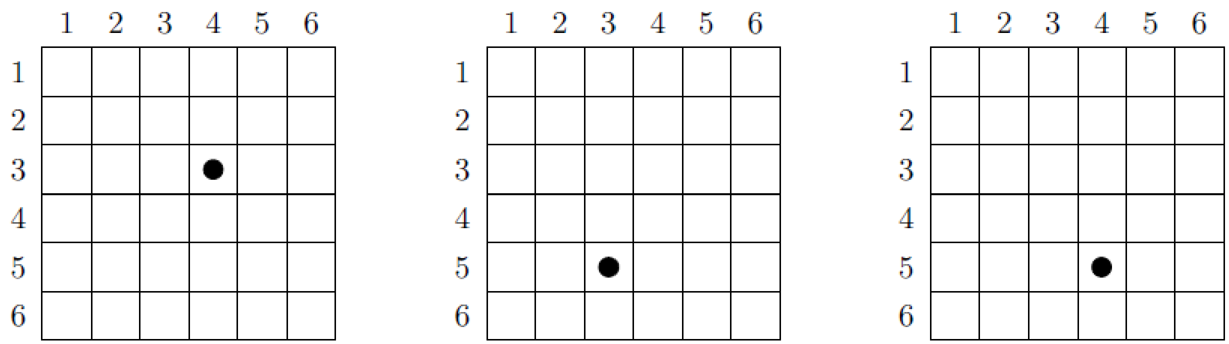
Example: 1 occurrence of *B* in *A*

Recall: Erosion

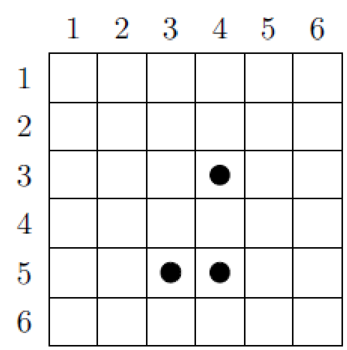
All occurrences of *B* in *A*



For each occurrences Mark center of *B*



Erosion: union of center of all occurrences of *B* in *A*



$A \ominus B$

Opening

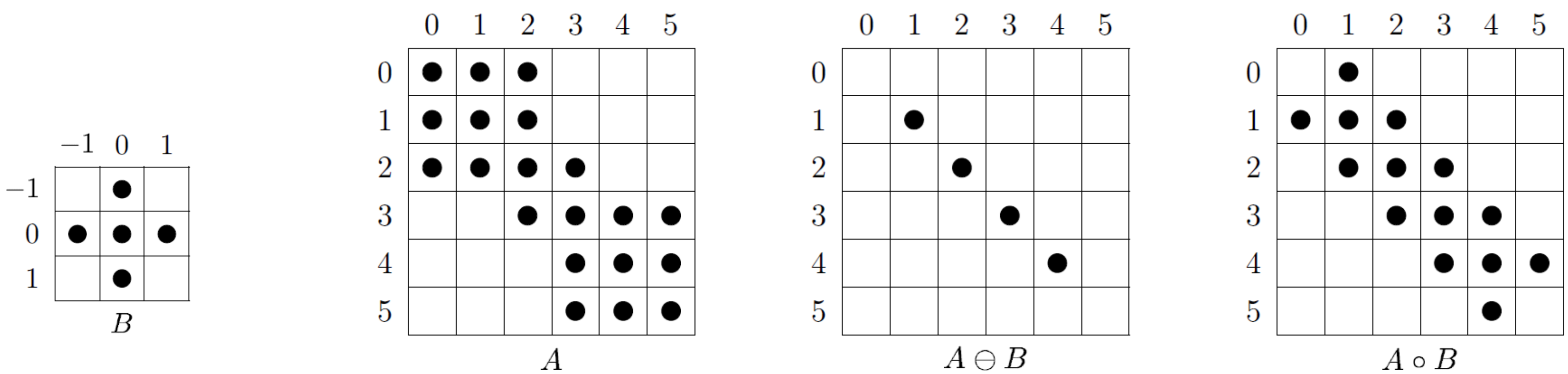
- Opening and closing: operations built on dilation and erosion
- Opening of *A* by structuring element *B*

$$A \circ B = (A \ominus B) \oplus B.$$

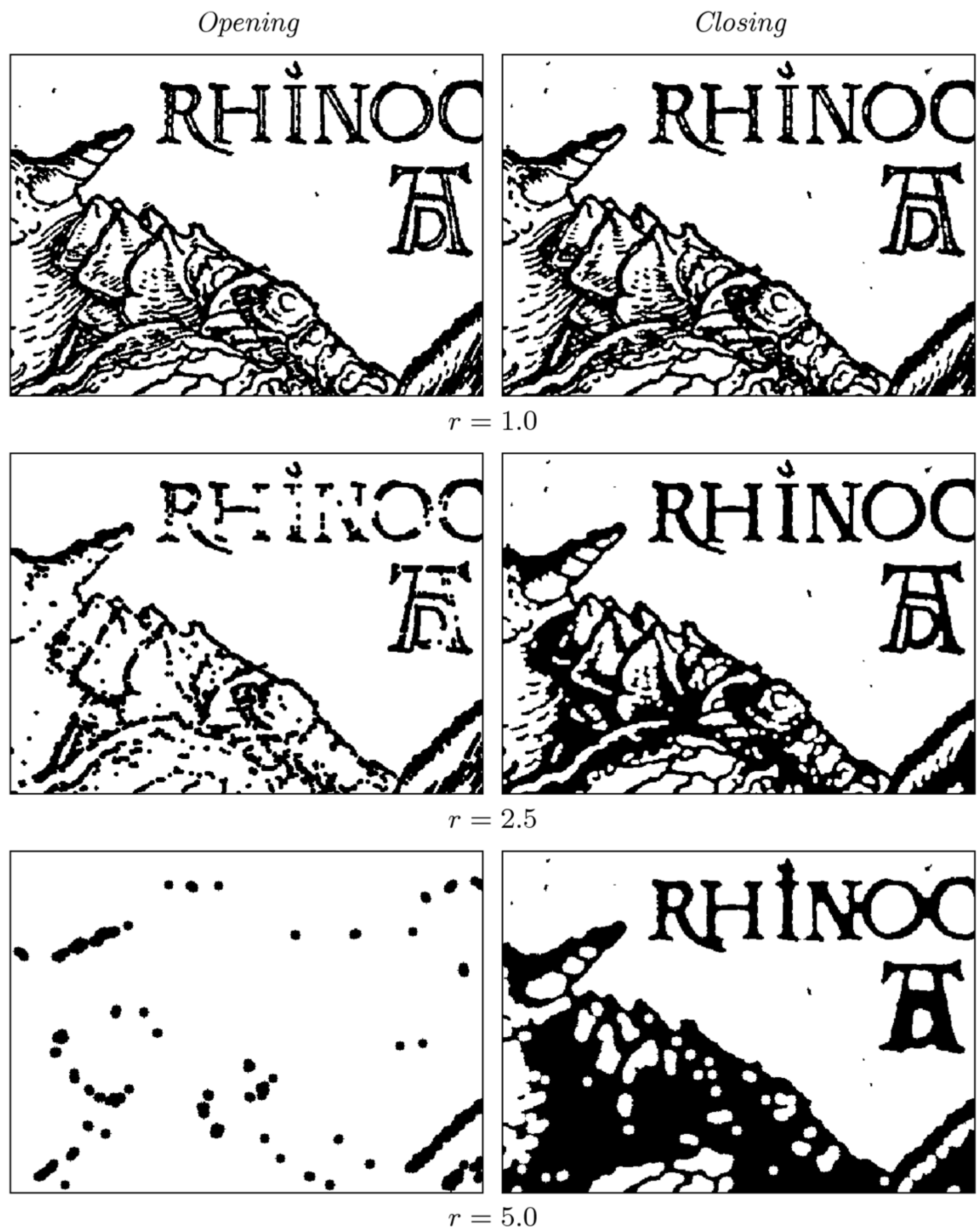
- i.e. opening = erosion followed by dilation. Alternatively

$$A \circ B = \cup \{B_w : B_w \subseteq A\}.$$

- i.e. Opening = union of all translations of *B* that fit in *A*
- **Note:** Opening includes all of B, erosion includes just (0,0) of *B*



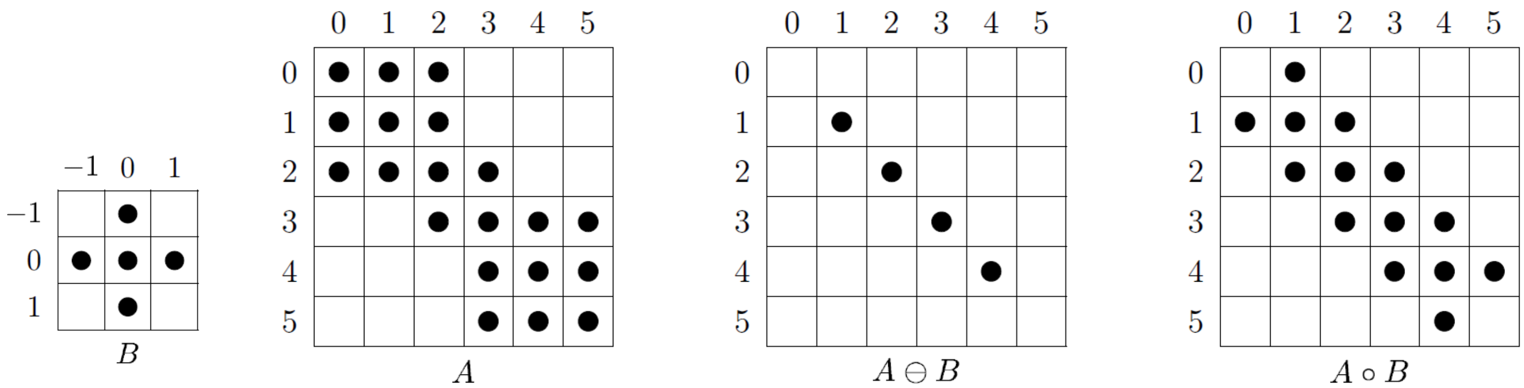
Opening



Binary opening and closing with disk-shaped Structuring elements of radius r = 1.0, 2.5, 5.0

- All foreground structures smaller than structuring element are eliminated by first step (erosion)
- Remaining structures smoothed by next step (dilation) then grown back to their original size

Properties of Opening



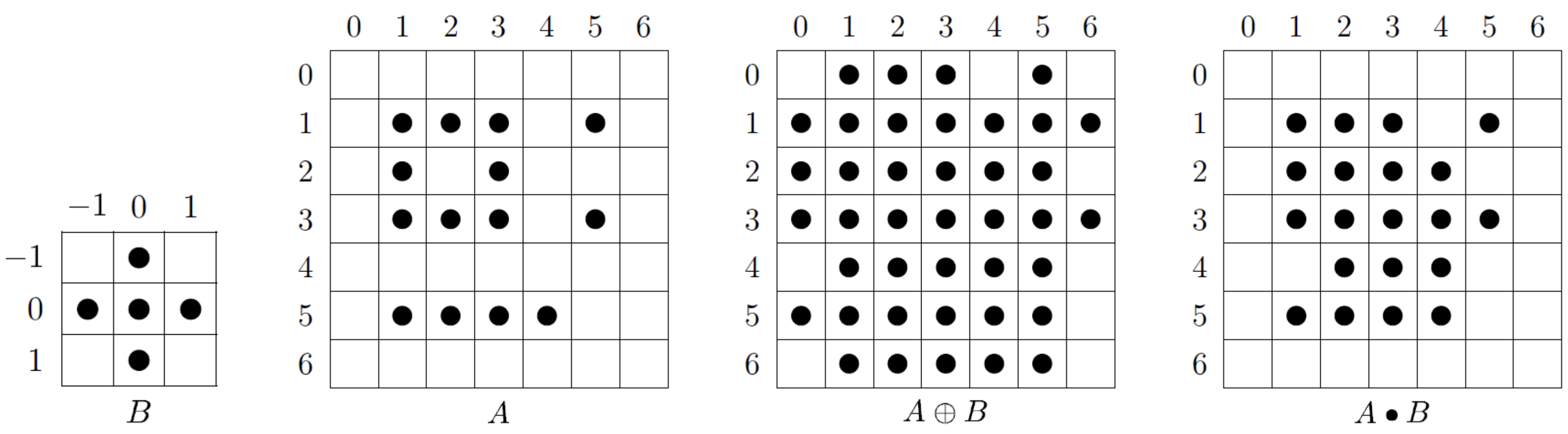
1. $(A \circ B) \subseteq A$. : Opening is subset of *A* (not the case with erosion)
2. $(A \circ B) \circ B = A \circ B$. : Can apply opening only once, also called **idempotence** (not the case with erosion
3. Subsets: If $A \subseteq C$, then $(A \circ B) \subseteq (C \circ B)$.
4. Opening tends to smooth an image, break narrow joins, and remove thin protrusions.

Closing

- Closing of *A* by structuring element *B*

$$A \bullet B = (A \oplus B) \ominus B.$$

- i.e. closing = dilation followed by erosion



Properties of Closing

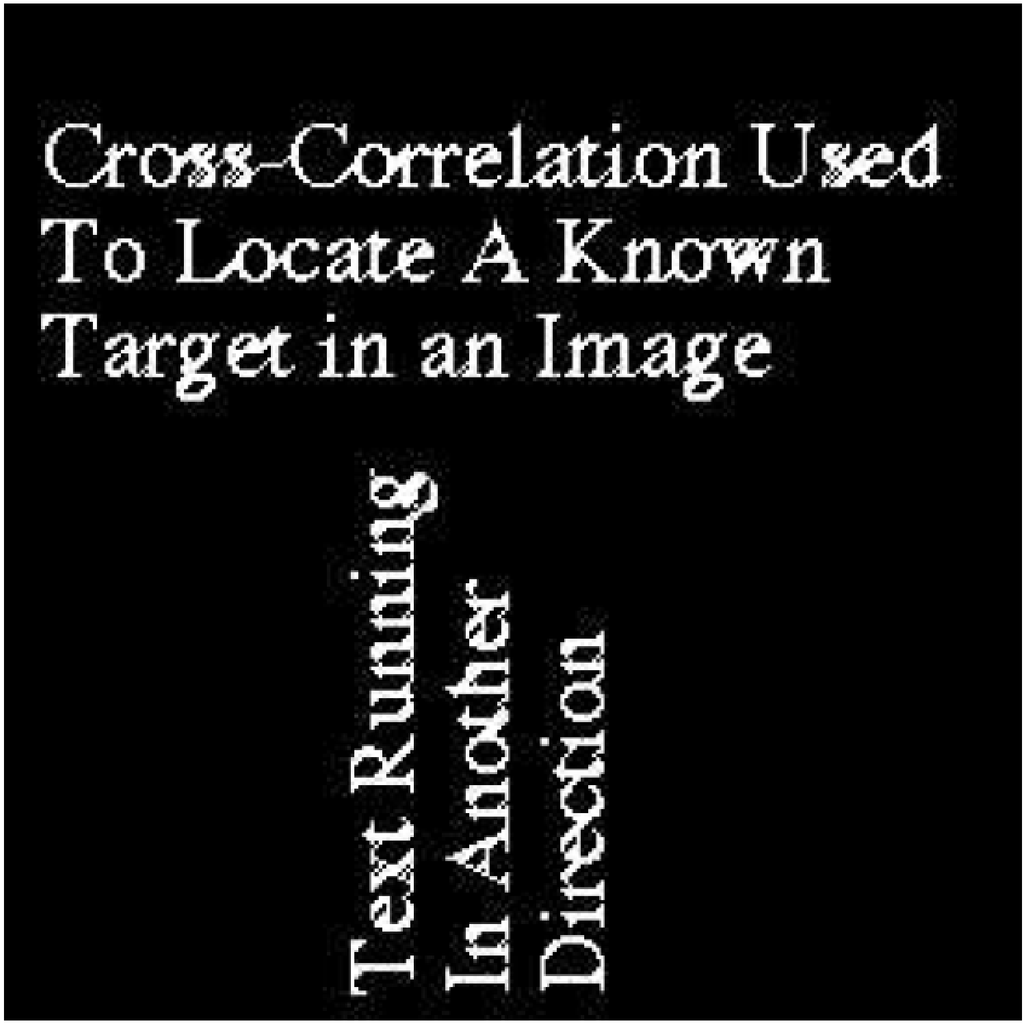
1. Subset: $A \subseteq (A \bullet B)$.

2. **Idempotence:** $(A \bullet B) \bullet B = A \bullet B$;

3. Also If $A \subseteq C$, then $(A \bullet B) \subseteq (C \bullet B)$.

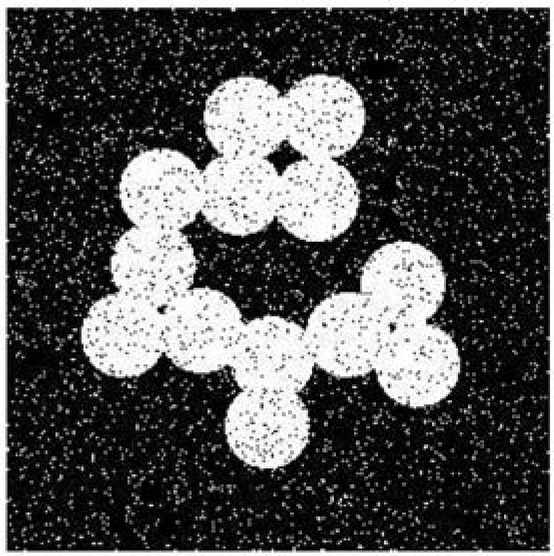
4. Closing tends to:
   a) Smooth an image
   b) Fuse narrow breaks and thin gulfs
   c) Eliminates small holes.

An Example of Closing



Noise Removal: Morphological Filtering

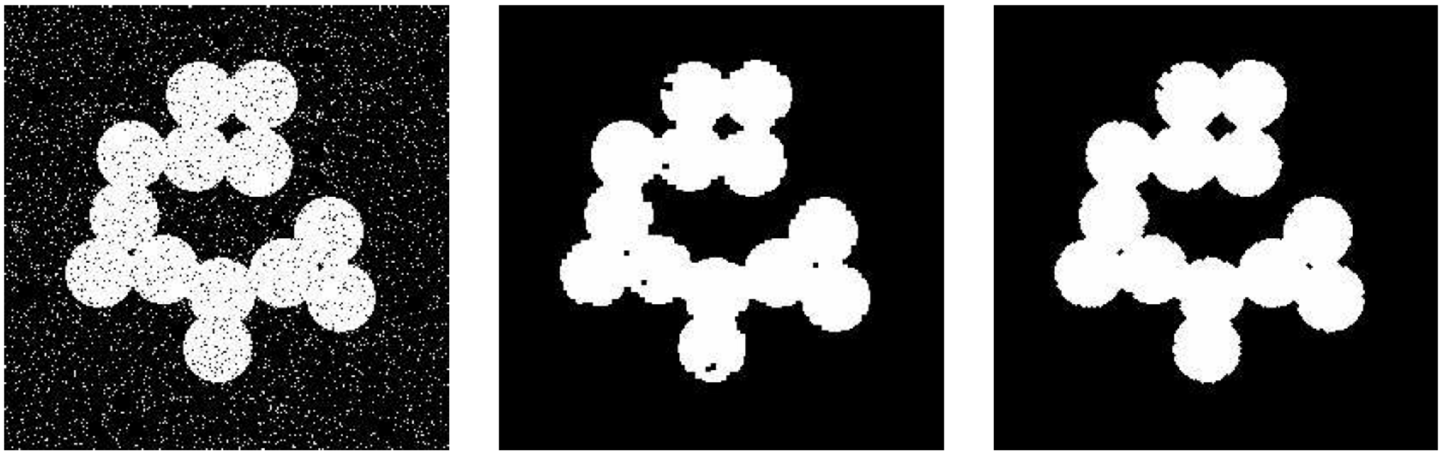
- Suppose *A* is image corrupted by impulse noise (some black, some white pixels, shown in (a) below)



(a)

- $A \ominus B$ removes single black pixels, but enlarges holes
- We can fill holes by dilating twice $((A \ominus B) \oplus B) \oplus B$.

Noise Removal: Morphological Filtering



(a) (b) (c)

(b) Filter once
(c) Filter Twice

- First dilation returns the holes to their original size
- Second dilation removes the holes but enlarges objects in image
- To reduce them to their correct size, perform a final erosion:

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

- Inner 2 operations = opening, Outer 2 operations = closing.
- This noise removal method = opening followed by closing

$$(A \circ B) \bullet B).$$

Relationship Between Opening and Closing

- Opening and closing are duals
  - i.e. Opening foreground = closing background, and vice versa

- Complement of an opening = the closing of a complement

$$\overline{A \bullet B} = \overline{A} \circ \hat{B}$$

- Complement of a closing = the opening of a complement.

$$\overline{A \circ B} = \overline{A} \bullet \hat{B}.$$

Grayscale Morphology

- Morphology operations can also be applied to grayscale images
- Just replace (OR, AND) with (MAX, MIN)
- Consequently, morphology operations defined for grayscale images can also operate on binary images (but not the other way around)
  - ImageJ has single implementation of morphological operations that works on binary and grayscale
- For color images, perform grayscale morphology operations on each color channel (RGB)
- For grayscale images, structuring element contains real values
- Values may be –ve or 0

Grayscale Morphology

- Elements in structuring element that have value 0 do contribute to result
- Design of structuring elements for grayscale morphology must distinguish between 0 and empty (don't care)

0	1	0
1	2	1
0	1	0

$\neq$

	1	
1	2	1
	1	

Grayscale Dilation

- **Grayscale dilation:** Max (value in filter *H* + image region)

$$(I \oplus H)(u,v) = \max_{(i,j)\in H} \{ I(u+i, v+j) + H(i,j) \}$$

1. Place filter *H* over region of image *I*

I

6	7	3	4
5	6	6	8
6	4	5	2
6	4	2	3

$\oplus$

H

1	1	1
1	2	1
1	1	1

$=$

$I \oplus H$

	8	9	
	7	9	

4. Place max value (8) at current filter origin

2. Add corresponding values (*I* + *H*)

$I + H$

7	8	4
6	8	7
7	5	6

max

3. Find max of all values (*I* + *H*) = 8

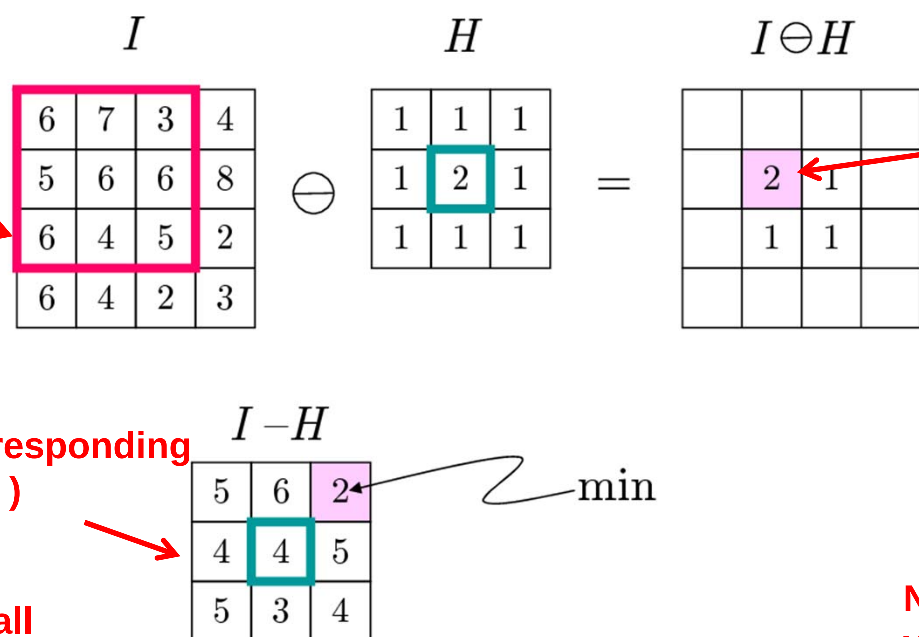
Note: Result may be negative value

Grayscale Erosion

- **Grayscale erosion:** Min (value in filter *H* + image region)

$$(I \ominus H)(u,v) = \min_{(i,j)\in H} \{ I(u+i, v+j) - H(i,j) \}$$

1. Place filter *H* over region of image *I*



4. Place min value (2) at current filter origin

2. Subtract corresponding values (*H* - *I*)

3. Find max of all values (*H* - *I*) = 2

Note: Result may be negative value

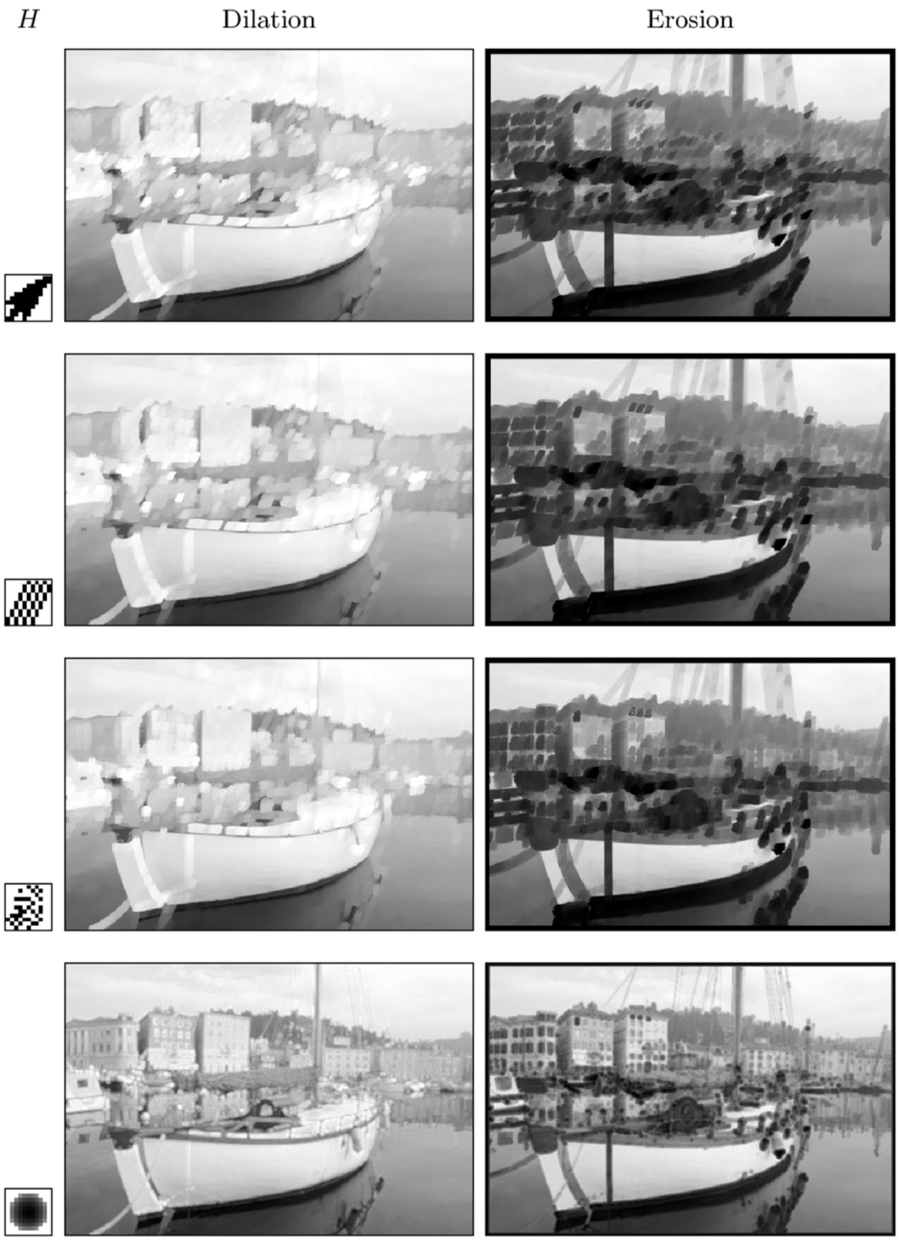
Grayscale Opening and Closing

- **Recall:** Opening = erosion then dilation:
- So we can implement grayscale opening as:
  - Grayscale erosion then grayscale dilation

- **Recall:** Closing = dilation then erosion:
- So we can implement grayscale erosion as:
  - Grayscale dilation then grayscale erosion

Grayscale Dilation and Erosion

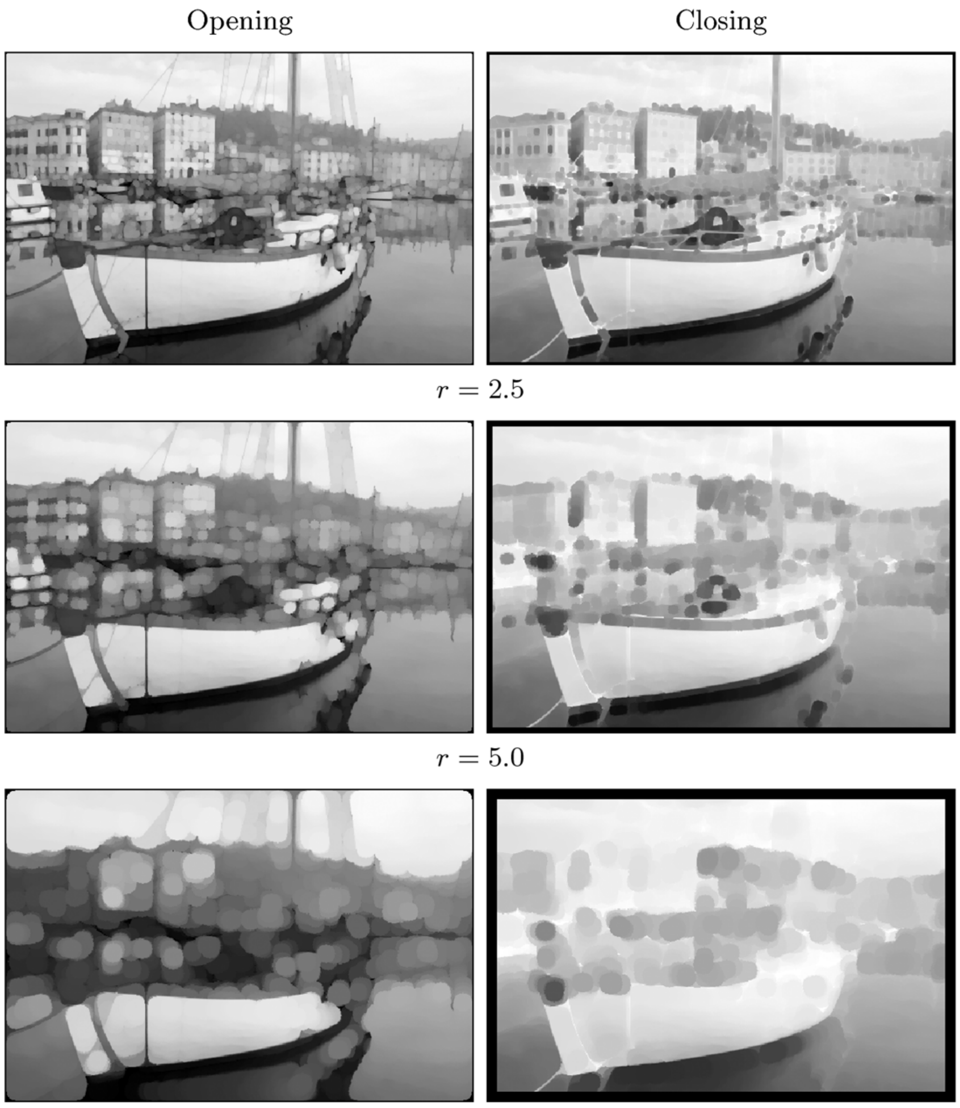
- Grayscale dilation and erosion with disk-shaped structuring elements of radius *r* = 2.5, 5.0, 10.0



Grayscale Dilation and Erosion

- Grayscale dilation and erosion with various free-form structuring elements

Grayscale Opening and Closing



- Grayscale opening and closing with disk-shaped structuring elements of radius *r* = 2.5, 5.0, 10.0

Implementing Morphological Filters

- Morphological operations implemented in ImageJ as methods of class **`ImageProcessor`**
  - **`dilate( )`**
  - **`erode( )`**
  - **`open( )`**
  - **`close( )`**
- The class **`BinaryProcessor`** offers these morphological methods
  - **`outline( )`**
  - **`skeletonize( )`**

Implementation of ImageJ dilate( )

```
...
void dilate(ImageProcessor I, int[][] H){
  //assume that the hot spot of H is at its center (ic,jc):
  int ic = (H[0].length-1)/2;
  int jc = (H.length-1)/2;

  //create a temporary (empty) image:
  ImageProcessor tmp
          = I.createProcessor(I.getWidth(),I.getHeight());

  for (int j=0; j<H.length; j++){
    for (int i=0; i<H[j].length; i++){
      if (H[j][i] > 0) { // this pixel is set
        //copy image into position (i-ic,j-jc):
        tmp.copyBits(I,i-ic,j-jc,Blitter.MAX);
      }
    }
  }
  //copy the temporary result back to original image
  I.copyBits(np,0,0,Blitter.COPY);
}
```

Center of filter H assumed to be at center

Create temporary copy of image

Perform dilation by copying shifted version of original into tmp

Replace original image destructively with tmp image

Implementation of ImageJ Erosion

- Erosion implementation can be derived from dilation
- **Recall:** Erosion is dilation of background
- So invert image, perform dilation, invert again

```
void erode(ImageProcessor I, int[][] H) {
  ip.invert();
  dilate(ip, reflect(H));
  ip.invert();
}
```

Implementation of Opening and Closing

- **Recall:** Opening = erosion then dilation:

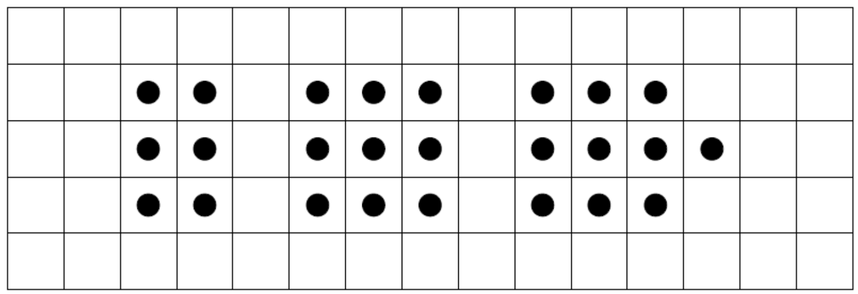
```
void open(ImageProcessor I, int[][] H) {
  erode(I,H);
  dilate(I,H);
}
```

- **Recall:** Closing = dilation then erosion:

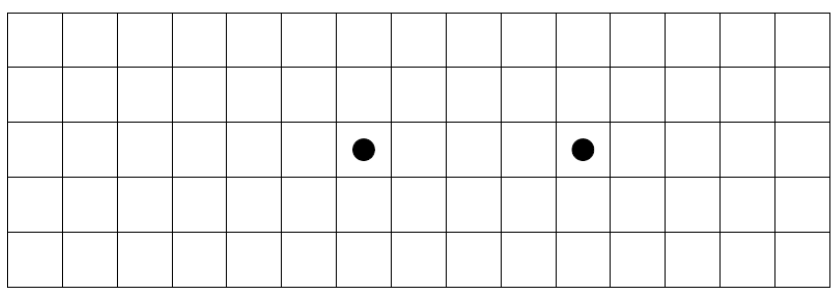
```
void close(ImageProcessor I, int[][] H) {
  dilate(I,H);
  erode(I,H);
}
```

Hit-or-Miss Transform

- Powerful method for finding shapes in images
- Can be defined in terms of erosion
- Suppose we want to locate 3x3 square shapes (in image center below)



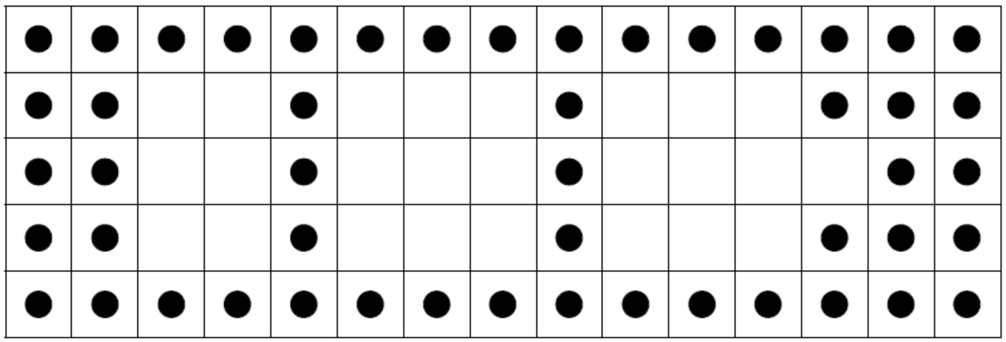
- If we perform an erosion $A \ominus B$ with *B* being the square element, result is:



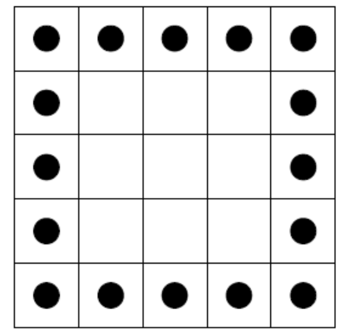
Hit or Miss Transform

- If we erode the complement of *A*, with a structuring element *C* that fits around 3x3 square

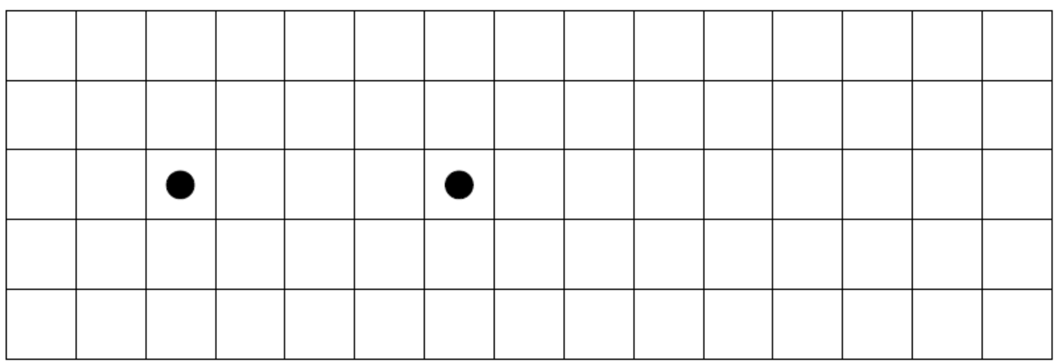
$\overline{A}$:



C:



- Result of $\overline{A} \ominus C$ is



- Intersection of 2 erosion operations produces 1 pixel at center of 3x3 square, which is what we want **(hit or miss transform)**

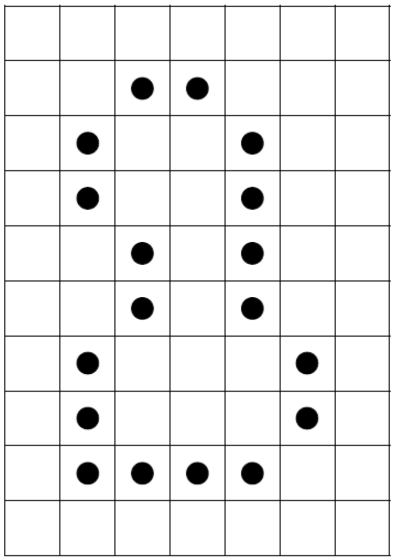
Hit-or-Miss Transform: Generalized

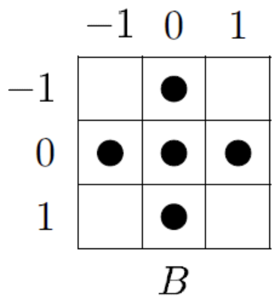
- If we are looking for a particular shape in an image, design 2 structuring elements:
  - B_1 which is same as shape we are looking for, and
  - B_2 which fits around the shape
  - We can then write B = (B_1, B_2)
- The hit-or-miss transform can be written as:

$$A \circledast B = (A \ominus B_1) \cap (\overline{A} \ominus B_2)$$

Morphological Algorithms: Region Filling

- Suppose an image has an 8-connected boundary





- Given a pixel **p** within the region, we want to fill region
- To do this, start with **p**, and dilate as many times as necessary with the cross-shaped structuring element **B**

Region Filling

- After each dilation, intersect with $\bar{A}$ before continuing
- We thus create the sequence:

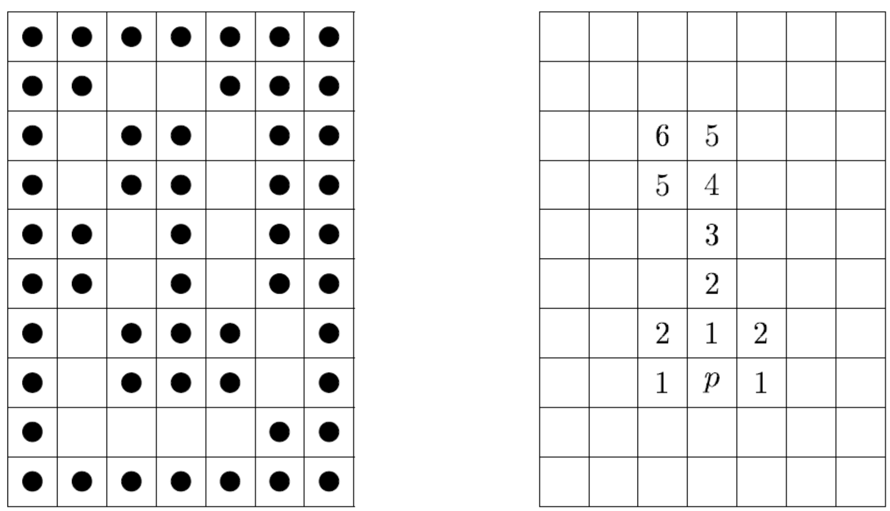
$$\{p\} = X_0, X_1, X_2, \ldots, X_k = X_{k+1}$$

for which

$$X_n = (X_{n-1} \oplus B) \cap \overline{A}.$$

- Finally $X_k \cup A$ is the filled region

$\overline{A}$:



Connected Components

- We use similar algorithm for connected components
  - Cross-shaped structuring element for 4-connected components
  - Square-shaped structuring element for 8-connected components
- To fill rest of component by creating sequence of sets

$$X_0 = \{p\}, X_1, X_2, \ldots$$

such that

$$X_n = (X_{n-1} \oplus B) \cap A$$

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

- Example:

●	●		●	●	
●	●	●		●	
			●	●	●
●	●	●			
●	●	●			
●	●	●			

2	1	2			
1	p	1			
2	1	2			

Using the cross

5	4		4	4	
5	4	3		3	
			2	3	4
1	1	1			
1	p	1			
1	1	1			

Using the square

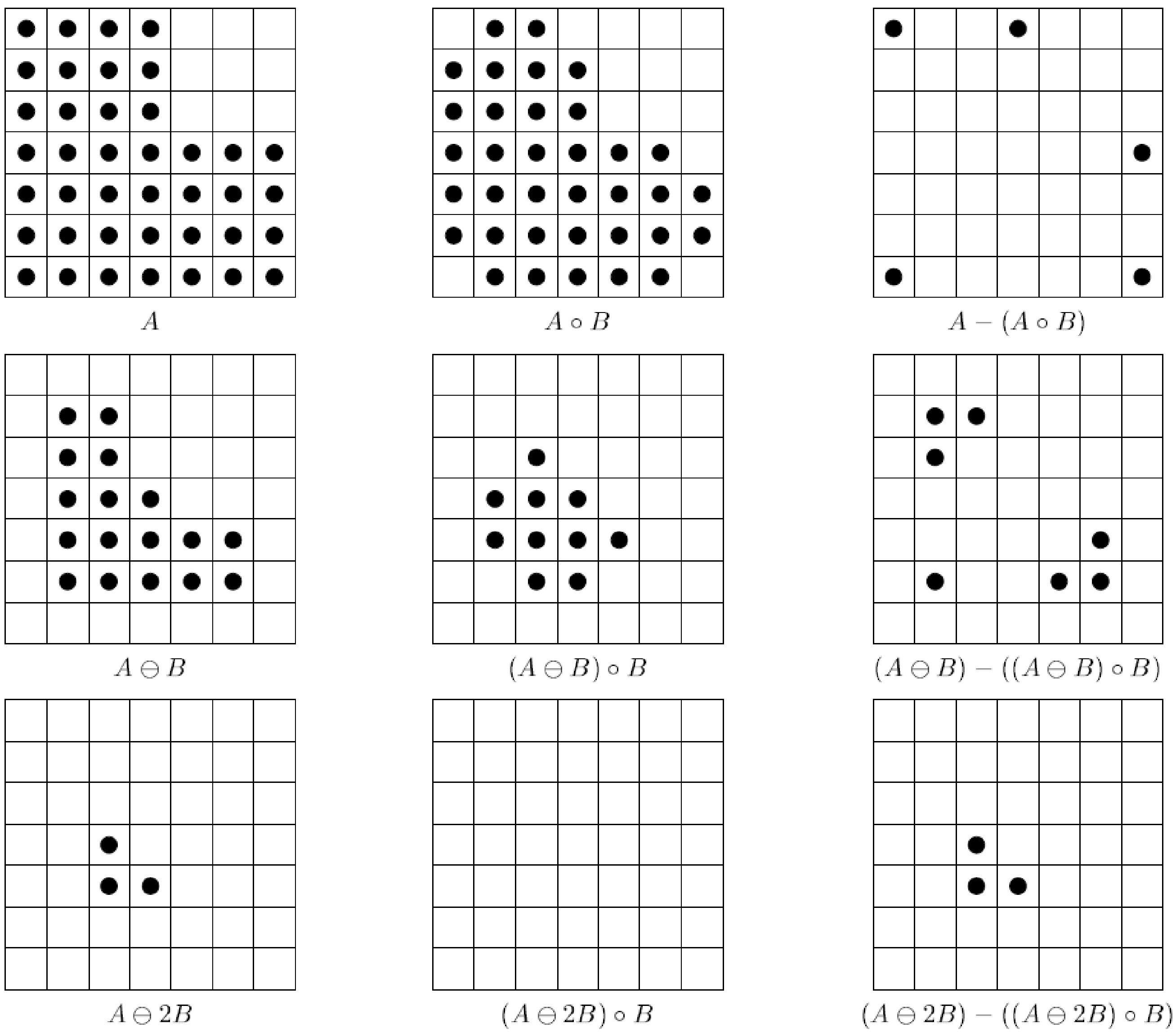
Skeletonization

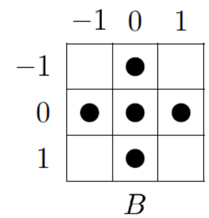
- Table of operations used to construct skeleton

Erosions	Openings	Set differences
A	$A \circ B$	$A - (A \circ B)$
$A \ominus B$	$(A \ominus B) \circ B$	$(A \ominus B) - ((A \ominus B) \circ B)$
$A \ominus 2B$	$(A \ominus 2B) \circ B$	$(A \ominus 2B) - ((A \ominus 2B) \circ B)$
$A \ominus 3B$	$(A \ominus 3B) \circ B$	$(A \ominus 3B) - ((A \ominus 3B) \circ B)$
$\vdots$	$\vdots$	$\vdots$
$A \ominus kB$	$(A \ominus kB) \circ B$	$(A \ominus kB) - ((A \ominus kB) \circ B)$

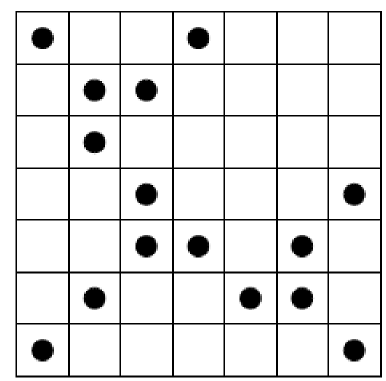
- Notation, sequence of *k* erosions with same structuring element: $A \ominus kB$
- Continue table until $(A \ominus kB) \circ B$ is empty
- Skeleton is union of all set differences

Skeletonization Example





Final skeletonization is union of all entries in 3rd column

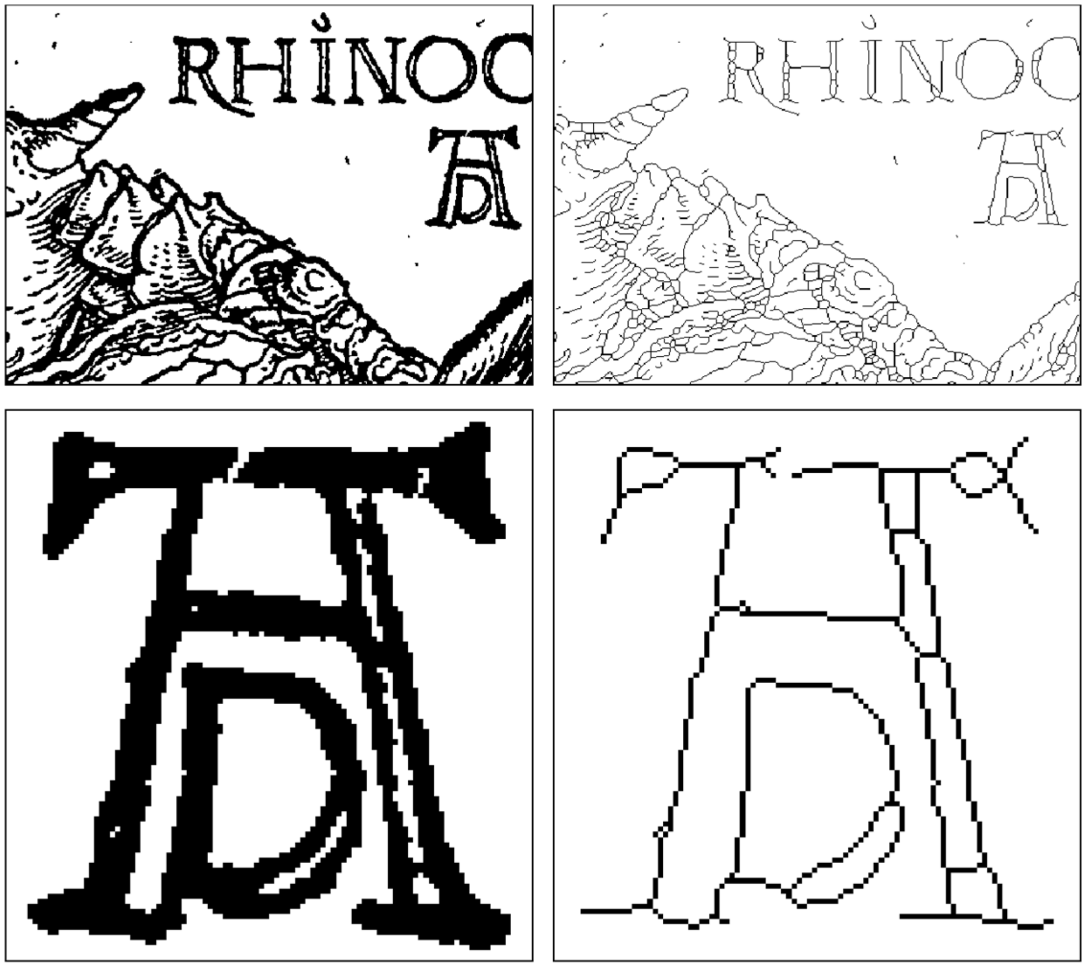


This method of skeletonization is called Lantuéjoul's method

Example: Thinning with Skeletonize( )

Original Image

Detail Image



(a) (b)

Results of thinning original Image

Results of thinning detail Image

References

- Wilhelm Burger and Mark J. Burge, Digital Image Processing, Springer, 2008
- Rutgers University, CS 334, Introduction to Imaging and Multimedia, Fall 2012
- Alasdair McAndrews, Introduction to Digital Image Processing with MATLAB, 2004

Computer Graphics (CS/ECE 545)
Lecture 7:
Regions in Binary Images (Part 1)

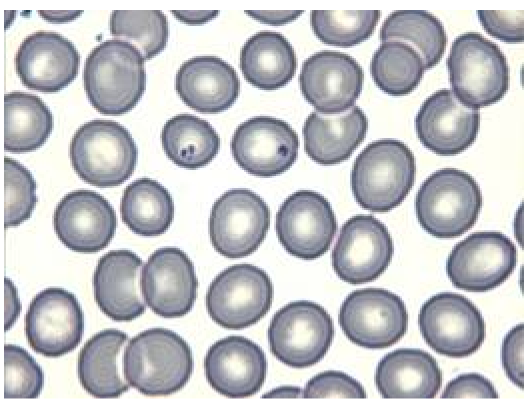
Prof Emmanuel Agu

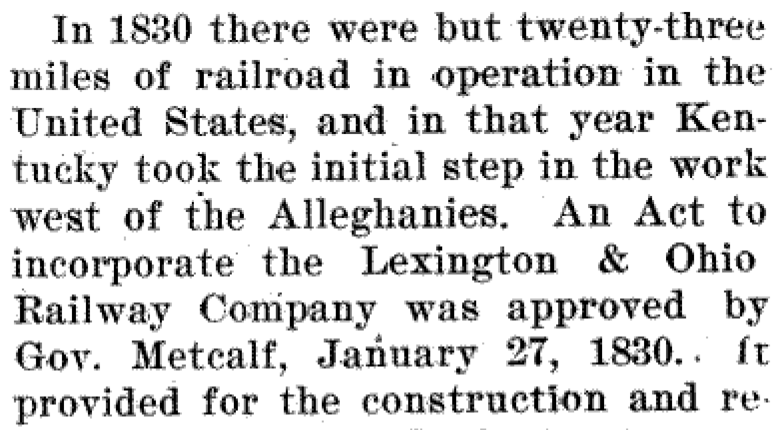
Computer Science Dept.

Worcester Polytechnic Institute (WPI)

Motivation

- High level vision task: recognize objects in flat black and white images:
  - Text on a page
  - Objects in a picture
  - Microscope images
- Image may be grayscale
  - Convert to black and white

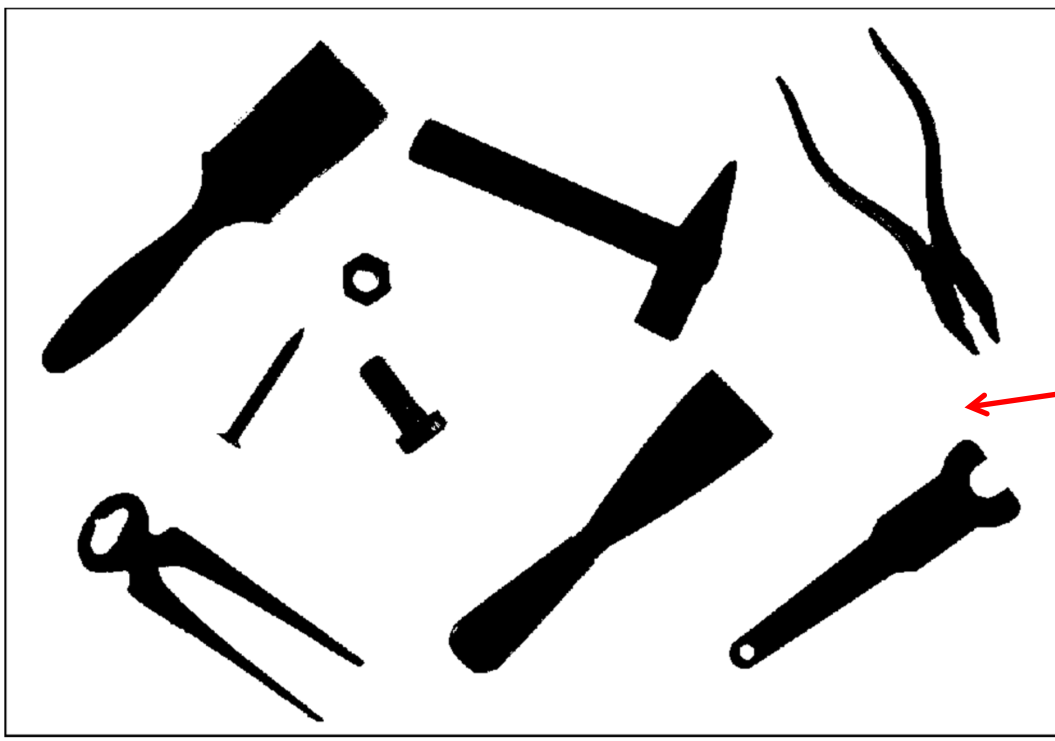




In 1830 there were but twenty-three miles of railroad in operation in the United States, and in that year Kentucky took the initial step in the work west of the Alleghanies. An Act to incorporate the Lexington & Ohio Railway Company was approved by Gov. Metcalf, January 27, 1830. It provided for the construction and re-

Motivation

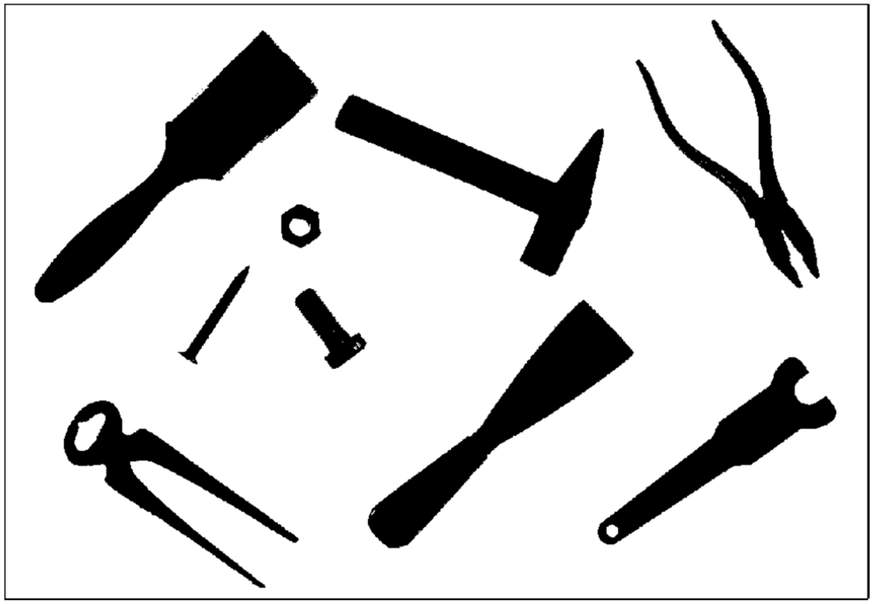
- Binary image: pixels can be black or white (foreground and background)
- Want to devise program that finds **number of objects** and **type of objects** in figure such as that below



Binary image
with 9 objects

Motivation

- Find objects by grouping together connected groups of pixels that belong to it
- Each object define a **binary region**
- After we find objects then what?
  - We can find out what objects are (**object types**) by comparing to models of different types of objects

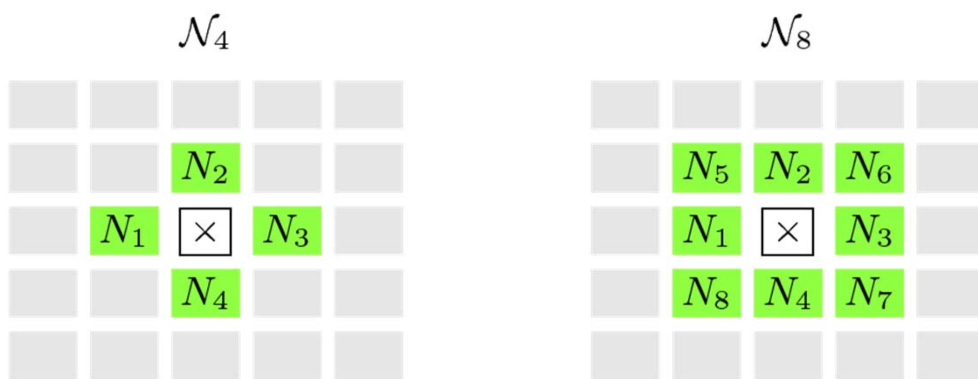


Finding Image Regions

- Most important tasks in searching for binary regions
  - Which pixels belong to which regions?
  - How many regions are in image?
  - Where are regions located?
- These tasks usually performed during **region labeling** (or **region coloring**)
- Find regions step by step, assign label to identify region
- 3 methods:
  - Flood filling
  - Sequential region labeling
  - Combine region labeling + contour finding

Finding Image Regions

- Must first decide whether we consider 4-connected (N_4) or 8-connected (N_8) pixels as neighbors



- Adopt following convention in binary images

$$I(u,v) = \begin{cases} 0 & \textit{background} \text{ pixel} \\ 1 & \textit{foreground} \text{ pixel} \\ 2, 3, \ldots & \text{region } \textit{label}. \end{cases}$$

Region Labeling with Flood Filling

- Searches for unmarked foreground pixel, then fill (visit and mark)
- 3 different versions:
  - Recursive
  - Depth-First
  - Breadth-First
- All 3 versions are called by the following region labeling algorithm

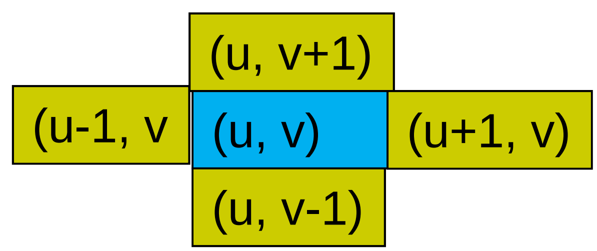
RegionLabeling(I)
    I: binary image (0 = *background*, 1 = *foreground*)
    The image I is labeled (destructively modified) and returned.

  Initialize $m \leftarrow 2$ (the value of the next label to be assigned).
  Iterate over all image coordinates (u, v).
    if $I(u, v) = 1$ **then**
      FloodFill(I, u, v, m) ▷ use any of the 3 versions below
      $m \leftarrow m + 1$.
  return the labeled image I.

Recursive Flood Filling

- Test each pixel recursively to find if each neighbor has *I(u,v)* = 1
- **Problem 1:** Each pixel can be tested up to 4 times (4 neighbors), inefficient!
- **Problem 2:** Stack can be exhausted quickly
  - Recursion depth is proportional to size of region
  - Thus, usage is limited to small images (approx < 200 x 200 pixels)

```
FloodFill(I, u, v, label)                                  ▷ Recursive Version
    if coordinate (u, v) is within image boundaries and I(u, v) = 1 then
        Set I(u, v) ← label
        FloodFill(I, u+1, v, label)
        FloodFill(I, u, v+1, label)
        FloodFill(I, u, v−1, label)
        FloodFill(I, u−1, v, label)
    return.
```



Depth-First Flood Filling

- Records unvisited elements in a stack
- Traverses tree of pixels **depth first**

```
FloodFill(I, u, v, label)                                ▷ Depth-First Version
    Create an empty stack S
    Put the seed coordinate ⟨u, v⟩ onto the stack: Push(S, ⟨u, v⟩)
    while S is not empty do
        Get the next coordinate from the top of the stack:
            ⟨x, y⟩ ← Pop(S)
        if coordinate (x, y) is within image boundaries and I(x, y) = 1
            then
            Set I(x, y) ← label
            Push(S, ⟨x+1, y⟩)
            Push(S, ⟨x, y+1⟩)
            Push(S, ⟨x, y−1⟩)
            Push(S, ⟨x−1, y⟩)
    return.
```

Breadth-First Flood Filling

- Similar to depth-first version
- Use queue to store unvisited elements instead of stack

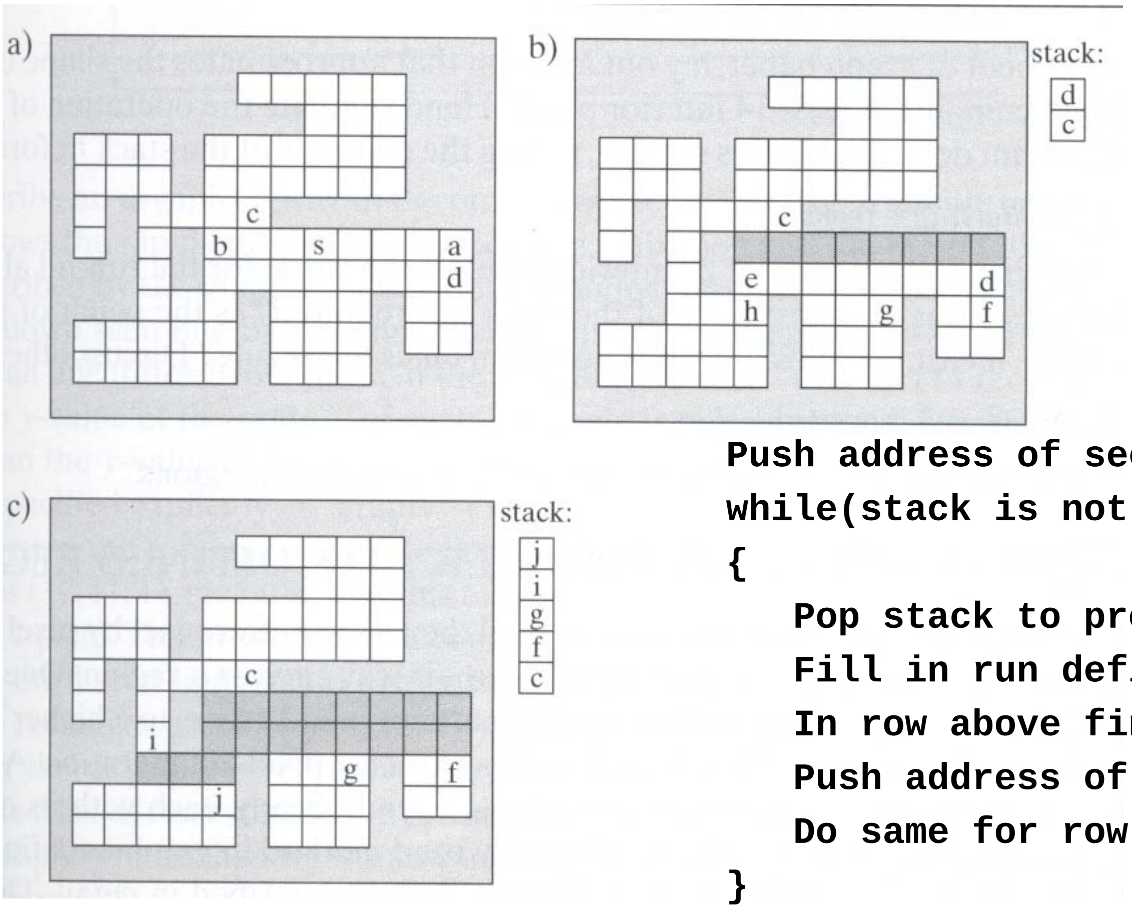
```
FloodFill(I, u, v, label)                          ▷ Breadth-First Version
    Create an empty queue Q
    Insert the seed coordinate ⟨u, v⟩ into the queue: Enqueue(Q, ⟨u, v⟩)
    while Q is not empty do
        Get the next coordinate from the front of the queue:
            ⟨x, y⟩ ← Dequeue(Q)
        if coordinate ⟨x, y⟩ is within image boundaries and I(x, y) = 1
            then
            Set I(x, y) ← label
            Enqueue(Q, ⟨x+1, y⟩)
            Enqueue(Q, ⟨x, y+1⟩)
            Enqueue(Q, ⟨x, y−1⟩)
            Enqueue(Q, ⟨x−1, y⟩)
    return.
```

Depth-First Flood-Filling

- Let's look at an implementation of depth-first flood filling
- **A run:** group of adjacent pixels lying on same scanline
- Fill runs(adjacent, on same scan line) of pixels

Region Filling Using Coherence

- Example: start at s, initial seed



Pseudocode:

```
Push address of seed pixel onto stack
while(stack is not empty)
{
   Pop stack to provide next seed
   Fill in run defined by seed
   In row above find reachable interior runs
   Push address of their rightmost pixels
   Do same for row below current run
}
```

Note: algorithm most efficient if there is **span coherence** (pixels on scanline have same value) and **scan-line coherence** (consecutive scanlines similar)

Java Code for Depth-First Flood Filling

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

```
void floodFill(ImageProcessor ip, int x, int y, int label) {
  Stack<Node> s = new Stack<Node>(); // stack
  s.push(new Node(x,y));
  while (!s.isEmpty()){
    Node n = s.pop();
    if ((n.x>=0) && (n.x<width) && (n.y>=0) && (n.y<height)
      && ip.getPixel(n.x,n.y)==1) {
      ip.putPixel(n.x,n.y,label);
      s.push(new Node(n.x+1,n.y));
      s.push(new Node(n.x,n.y+1));
      s.push(new Node(n.x,n.y-1));
      s.push(new Node(n.x-1,n.y));
    }
  }
}
```

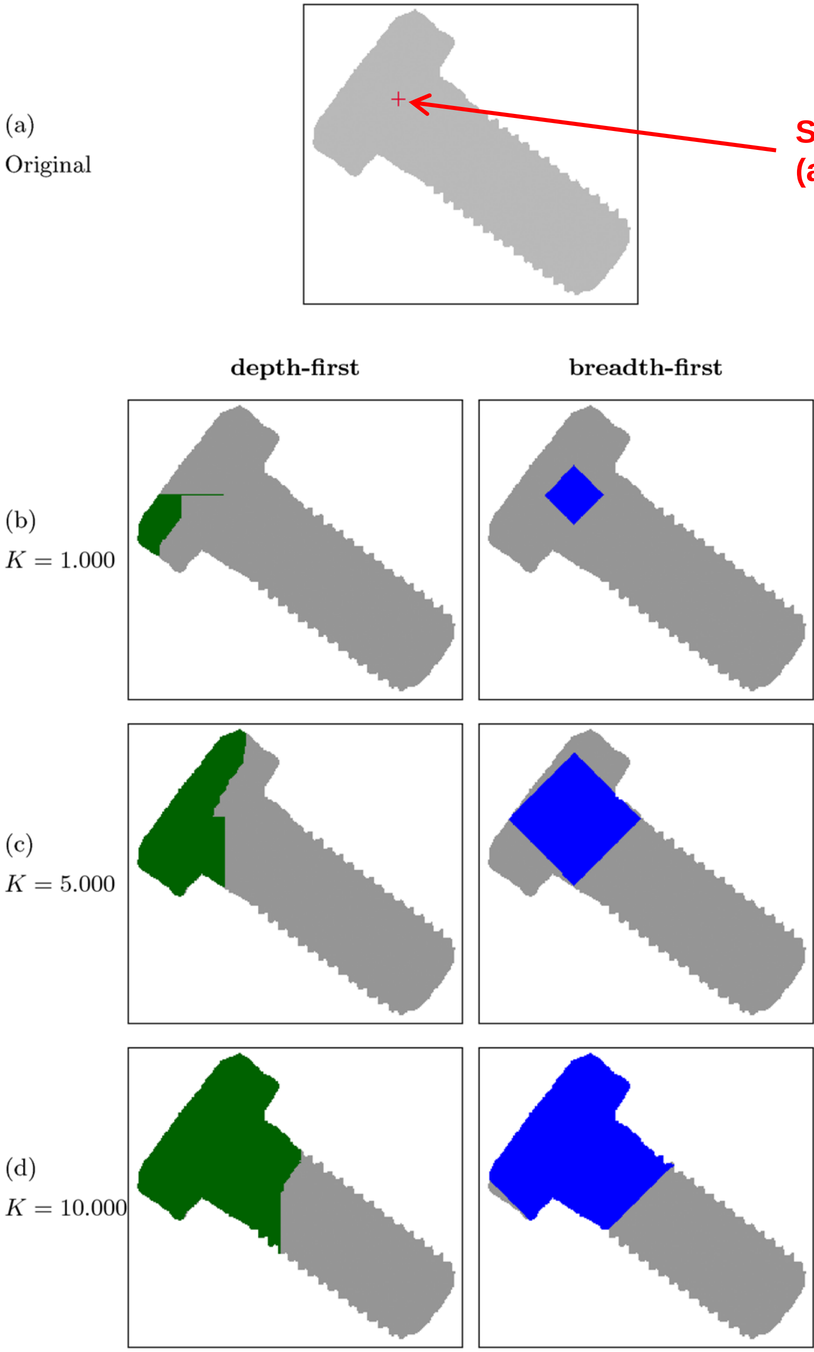
Uses `push( )`, `pop( )`
`isEmpty( )` methods
Of Java class `Stack`

Java Code for Breadth-First Flood Filling

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

```
void floodFill(ImageProcessor ip, int x, int y, int label) {
  LinkedList<Node> q = new LinkedList<Node>(); // queue
  q.addFirst(new Node(x,y));
  while (!q.isEmpty()) {
    Node n = q.removeLast();
    if ((n.x>=0) && (n.x<width) && (n.y>=0) && (n.y<height)
      && ip.getPixel(n.x,n.y)==1) {
      ip.putPixel(n.x,n.y,label);
      q.addFirst(new Node(n.x+1,n.y));
      q.addFirst(new Node(n.x,n.y+1));
      q.addFirst(new Node(n.x,n.y-1));
      q.addFirst(new Node(n.x-1,n.y));
    }
  }
}
```

Uses Java class `LinkedList`
with access methods
`addFirst( )`for ENQUEUE()
`removeLast( )`for DEQUEUE()



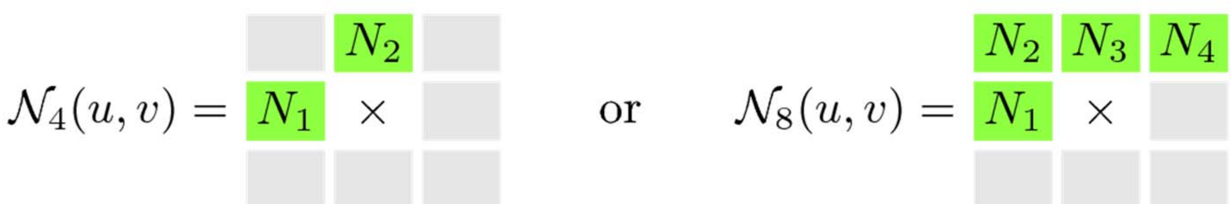
Starting point (arbitrary)

Intermediate results after
K = 1000, 5,000 and 10,000 iterations

Comparing Depth-First Vs Breadth-First Flood Filling

Sequential Region Labeling

- 2 steps:
  1. Preliminary labeling of image regions
  2. Resolving cases where more than one label occurs (been previously labeled)
- Even though algorithm is complex (especially 2nd stage), it is preferred because it has lower memory requirements
- First step: preliminary labeling
- Check following pixels depending on if we consider 4-connected or 8-connected neighbors



Preliminary Labeling: Propagating Labels

- Consider the following image:

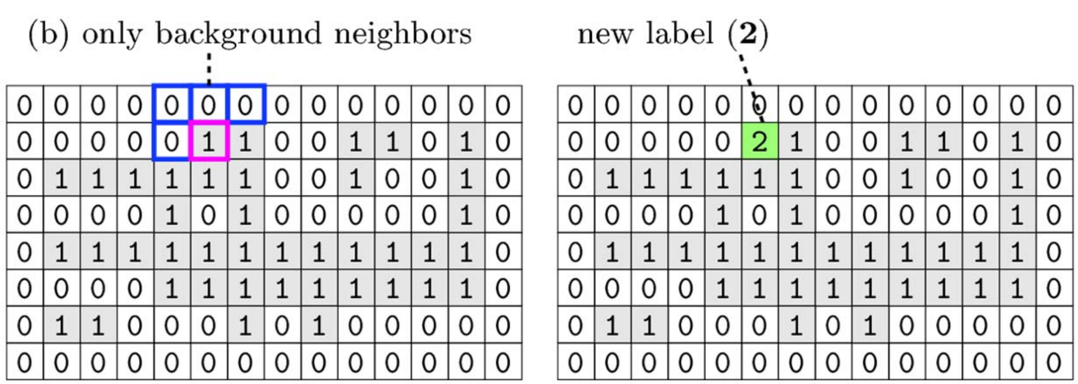
(a)

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

0 Background

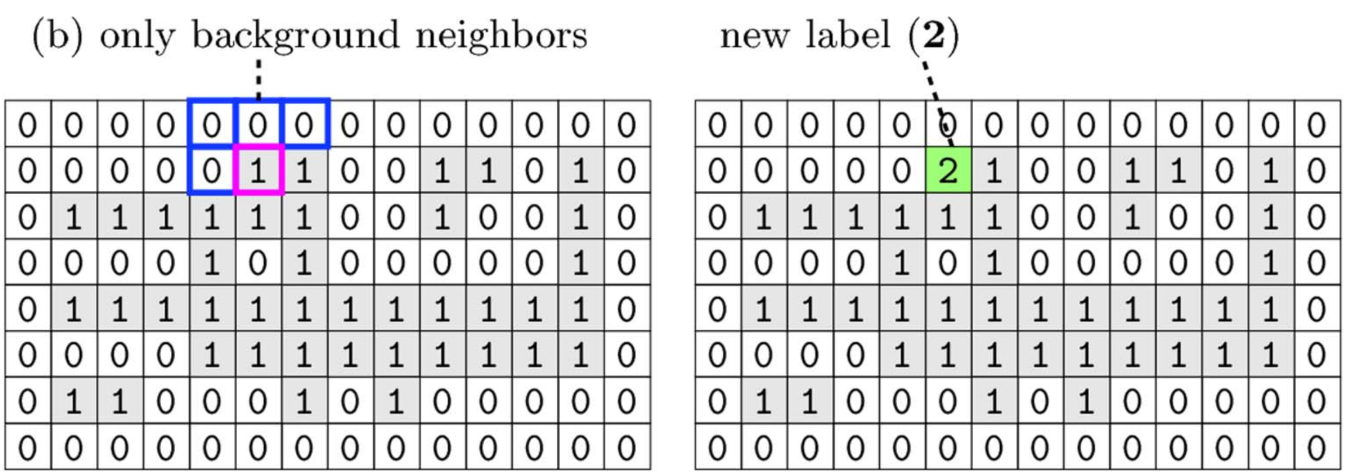
1 Foreground

- Neighboring pixels outside image considered part of background
- Slide Neighborhood region *N(u,v)* horizontally then vertically starting from top left corner



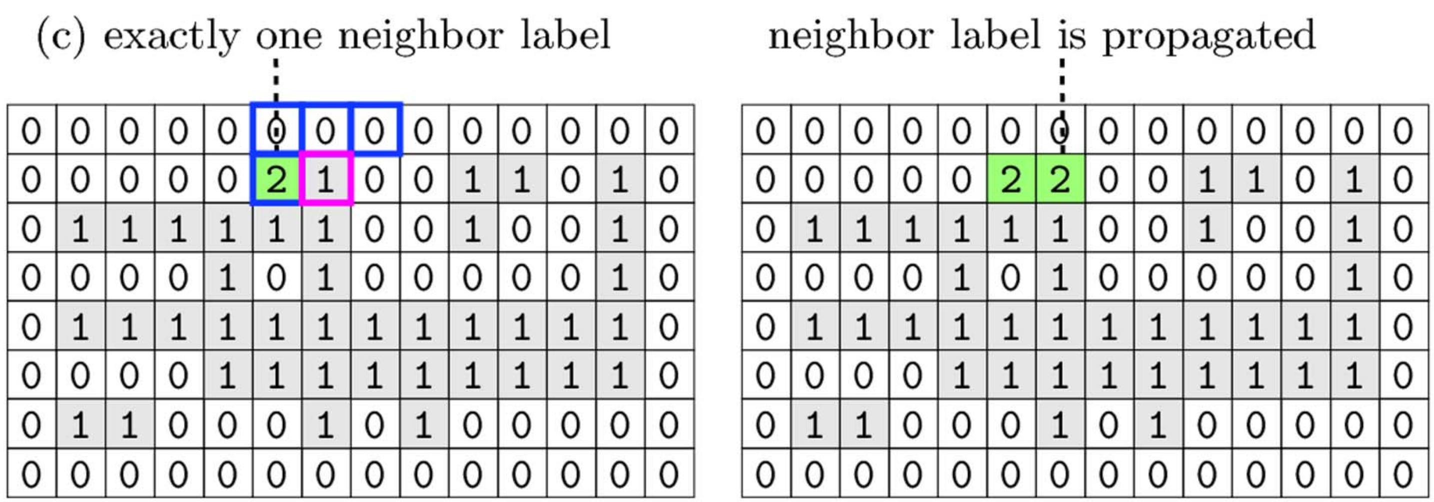
Preliminary Labeling: Propagating Labels

- First foreground pixel **[1]** is found
- All neighbors in *N(u,v)* are background pixels **[0]**
- Assign pixel the first label **[2]**



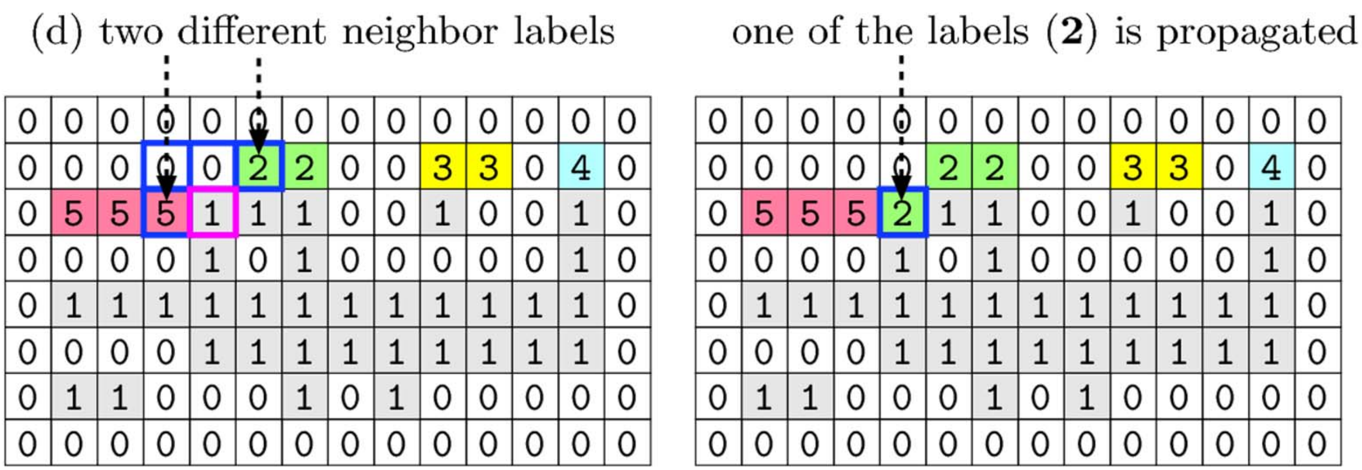
Preliminary Labeling: Propagating Labels

- In next step, exactly on neighbor in *N(u,v)* marked with label 2, so propagate this value **[2]**



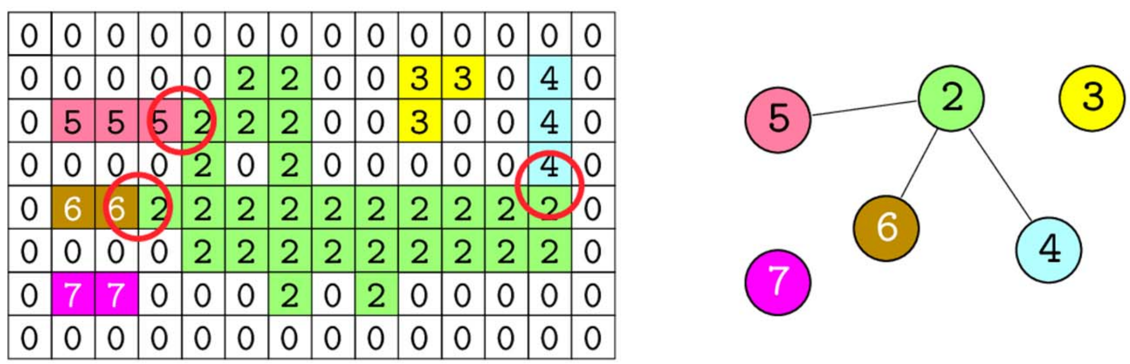
Preliminary Labeling: Propagating Labels

- Continue checking pixels as above
- At step below, there are **two** neighboring pixels and they have differing labels (**2** and **5**)
- One of these values is propagated (2 in this case), and collision <**2**,**5**> is registered



Preliminary Labeling: Label Collisions

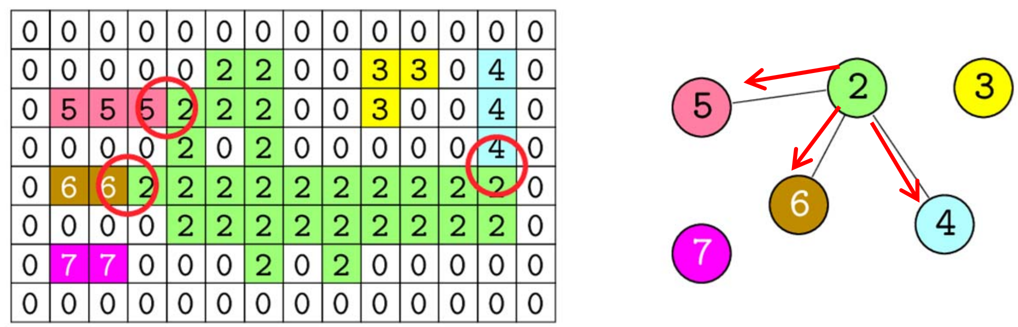
- At the end of labeling step
  - All foreground pixels have been provisionally marked
  - All collisions between labels (red circles) have been registered
  - Labels and collisions correspond to edges of undirected graph



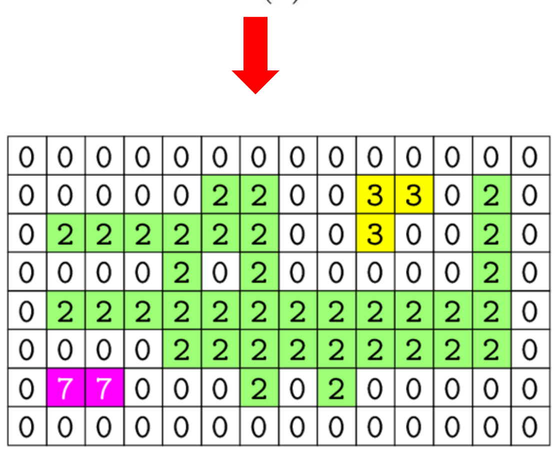
(a) (b)

Resolving Collisions

- Once all distinct labels within single region have been collected, assign labels of all pixels in region to be the same (e.g. assign all labels to have the smallest original label. E.g. **[2]**



(a) (b)



Sequential Region Labeling

SequentialLabeling(I)
    I: binary image ($0 =$ *background*, $1 =$ *foreground*)
    The image I is labeled (destructively modified) and returned.

Pass 1—Assign Initial Labels:
    Initialize $m \leftarrow 2$ (the value of the next label to be assigned).
    Create an empty set $\mathcal{C}$ to hold the collisions: $\mathcal{C} \leftarrow \{\}$.
    for $v \leftarrow 0 \ldots H-1$ **do**   ▷ $H =$ height of image I
        for $u \leftarrow 0 \ldots W-1$ **do**   ▷ $W =$ width of image I
            if $I(u,v) = 1$ **then** do one of:
                if all neighbors of (u,v) are background pixels (all $n_i = 0$) **then**
                    $I(u,v) \leftarrow m$.
                    $m \leftarrow m + 1$.
                else if exactly *one* of the neighbors has a label value $n_k > 1$ **then**
                    set $I(u,v) \leftarrow n_k$
                else if *several* neighbors of (u,v) have label values $n_j > 1$ **then**
                    Select one of them as the new label:
                        $I(u,v) \leftarrow k \in \{n_j\}$.
                    for all other neighbors of $u,v)$ with label values $n_i > 1$ and $n_i \neq k$ **do**
                        Create a new label collision $c_i = \langle n_i, k \rangle$.
                        Record the collision: $\mathcal{C} \leftarrow \mathcal{C} \cup \{c_i\}$.

Remark: The image I now contains label values $0, 2, \ldots m-1$.

← **Preliminary labeling**

Sequential Region Labeling

PASS 2—RESOLVE LABEL COLLISIONS:

Let $\mathcal{L} = \{2, 3, \ldots m-1\}$ be the set of preliminary region labels.

Create a partitioning of $\mathcal{L}$ as a *vector of sets*, one set for each label value: $\mathcal{R} \leftarrow [\mathcal{R}_2, \mathcal{R}_3, \ldots, \mathcal{R}_{m-1}] = [\{2\}, \{3\}, \{4\}, \ldots, \{m-1\}]$, so $\mathcal{R}_i = \{i\}$ for all $i \in \mathcal{L}$.

for all collisions $\langle a, b \rangle \in \mathcal{C}$ **do**

  Find in $\mathcal{R}$ the sets $\mathcal{R}_a$, $\mathcal{R}_b$ containing the labels a, b, resp.:

    $\mathcal{R}_a \leftarrow$ the set that currently contains label a

    $\mathcal{R}_b \leftarrow$ the set that currently contains label b

  if $\mathcal{R}_a \neq \mathcal{R}_b$ (a and b are contained in different sets) **then**

    Merge sets $\mathcal{R}_a$ and $\mathcal{R}_b$ by moving all elements of $\mathcal{R}_b$ to $\mathcal{R}_a$:

      $\mathcal{R}_a \leftarrow \mathcal{R}_a \cup \mathcal{R}_b$

      $\mathcal{R}_b \leftarrow \{\}$

Remark: All *equivalent* label values (i. e., all labels of pixels in the same region) are now contained in the same set $\mathcal{R}_i$ within $\mathcal{R}$.

Resolve label collisions

PASS 3—RELABEL THE IMAGE:

Iterate through all image pixels (u, v):

  if $I(u, v) > 1$ **then**

    Find the set $\mathcal{R}_i$ in $\mathcal{R}$ that contains label $I(u, v)$.

    Choose one unique representative element k from the set $\mathcal{R}_i$ (e. g., the minimum value, $k = \min(\mathcal{R}_i)$).

    Replace the image label: $I(u, v) \leftarrow k$.

return the labeled image I.

Relabel Image

References

- Wilhelm Burger and Mark J. Burge, Digital Image Processing, Springer, 2008
- Rutgers University, CS 334, Introduction to Imaging and Multimedia, Fall 2012