Computer Graphics

Scan Conversion

Dr. Mousumi Dutt

CSE, STCET

Introduction

- Assumption: raster display
- How the scene is displayed?
  - by loading the pixel arrays into the frame buffer
  - by scan converting the basic geometric-structure specifications into pixel patterns
- Scene Description
  - In terms of the basic geometric structures (provided by graphics package), referred to as *output primitives*
  - Group sets of output primitives into more complex structures
  - Each output primitive is specified with input coordinate data and other information about the way that object is to be displayed
  - Simplest geometric components: **Points and Straight line segments**
  - Additional output primitives: **circles and other conic sections, quadric surfaces, spline curves and surfaces, polygon color areas, and character strings**

Point Plotting

Converting a single coordinate position furnished by an application program into appropriate operations for the output device in use

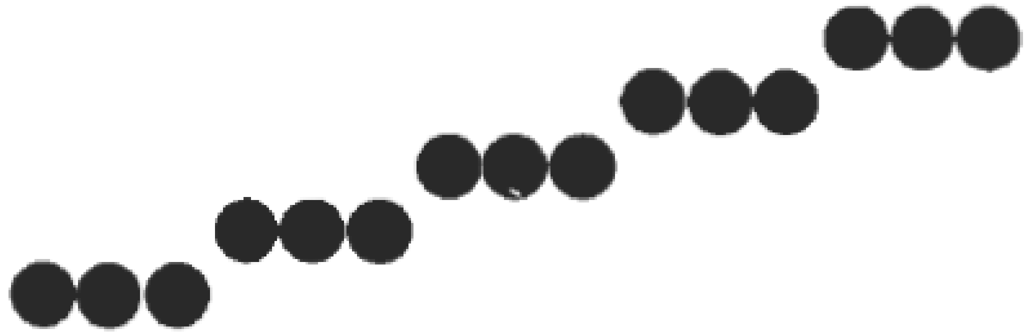
- CRT: the electron beam is turned on to illuminate the screen phosphor at the selected location
- A random-scan (vector) system:
  - stores point-plotting instructions in the display list
  - coordinate values in these instructions are converted to deflection voltages
  - that position the electron beam at the screen locations to be plotted during each refresh cycle
- Black and-white raster system:
  - setting the bit value corresponding to a specified screen position within the frame buffer to 1
  - the electron beam sweeps across each horizontal scan line
  - it emits a burst of electrons (plots a point) whenever a value of 1 in the frame buffer
- RGB system: The frame buffer is loaded with the color codes for the intensities that are to be displayed at the screen pixel positions

Line Plotting

- By calculating intermediate positions along the line path between two specified endpoint positions
- Vector pen plotter or a random-scan display: Linearly varying horizontal and vertical deflection voltages are generated that are proportional to the required changes in the x and y directions to produce the smooth line
- Digital devices: by plotting discrete points between the two endpoints
  - Screen position is approximated
  - The line color (intensity) is then loaded into the frame buffer at the corresponding pixel coordinates
  - Reading from the frame buffer, the video controller "plots" the screen pixels
  - The rounding of coordinate values to integers causes lines to be displayed with a stairstep appearance ***("the jaggies")***

Line Plotting: jaggies

- Noticeable on systems with low resolution
- Improve their appearance somewhat by displaying them on high-resolution systems
- More effective techniques for smoothing raster lines are based on adjusting pixel intensities along the line paths



Antialiasing

- **Jagged or stairstep appearance:** as the sampling process digitizes coordinate points on an object to discrete integer pixel positions
- This distortion of information due to low-frequency sampling (undersampling) is called ***aliasing***
- Applying ***antialiasing*** methods compensate for the undersampling process

Antialiasing

- Set the sampling frequency to at least twice that of the highest frequency occurring in the object, referred to as the Nyquist sampling frequency (or Nyquist sampling rate): $f_s = 2f_{max}$
- The sampling interval should be no larger than one-half the cycle interval (called the Nyquist sampling interval)
- For x-interval sampling, the Nyquist sampling interval is $\Delta x_s = \frac{\Delta x_{cycle}}{2}$ where $\Delta x_{cycle} = 1/f_{max}$
- Unless hardware technology is developed to handle arbitrarily large frame buffers, increased screen resolution is not a complete solution to the aliasing problem

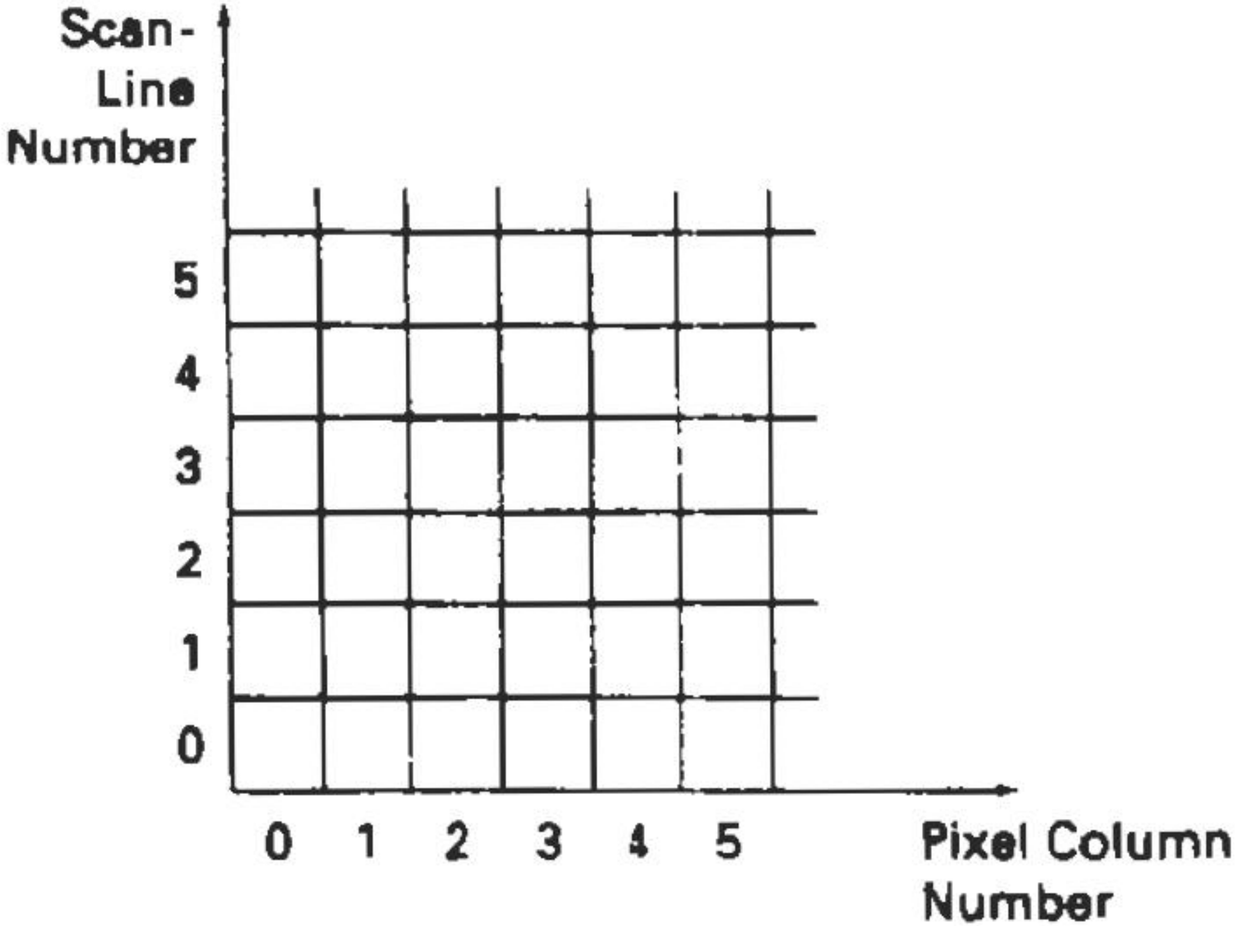
Antialiasing

- To increase sampling rate by treating the screen as if it were covered with a finer grid than is actually available
- Use multiple sample points across this finer grid to determine an appropriate intensity level for each screen pixel
- This technique of sampling object characteristics at a high resolution and displaying the results at a lower resolution is called *supersampling* (or *postfiltering*, since the general method involves computing intensities, it subpixel grid positions, then combining the results to obtain the pixel intensities)
- By supersampling, we obtain intensity information from multiple points that contribute to the overall intensity of a pixel

Antialiasing

- **Area sampling or prefiltering**
  - To determine pixel intensity by calculating the areas of overlap of each pixel with the objects to be displayed
  - The intensity of the pixel as a whole is determined without calculating subpixel intensities
- **Pixel Phasing:** Raster objects can also be antialiased by shifting the display location of pixel areas
- applied by "micropositioning” the electron beam in relation to object geometry.

Pixel positions referenced by scanline number and column number



Line Drawing Algorithms

- The Cartesian slope-intercept equation for a straight line is y=m*x+b
  - m= slope; b= y-intercept
- Given that the two endpoints of a h e segment are specified at positions *(x_1, y_1) and (x_2, y_2)*

$$m = \frac{y_2 - y_1}{x_2 - x_1};\ b = y_1 - mx_1$$

$|\mathrm{m}| < 1, \Delta\mathrm{x}$ can be set proportional to a small horizontal deflection voltage,

$$m = \frac{\Delta y}{\Delta x}$$

Vertical deflection is set proportional to $\Delta\mathrm{y}$ as calculated

$|\mathrm{m}| > 1, \Delta\mathrm{y}$ can be set proportional to a small horizontal deflection voltage,

$$\Delta x = \frac{\Delta y}{m}$$

Vertical deflection is set proportional to $\Delta\mathrm{x}$ as calculated

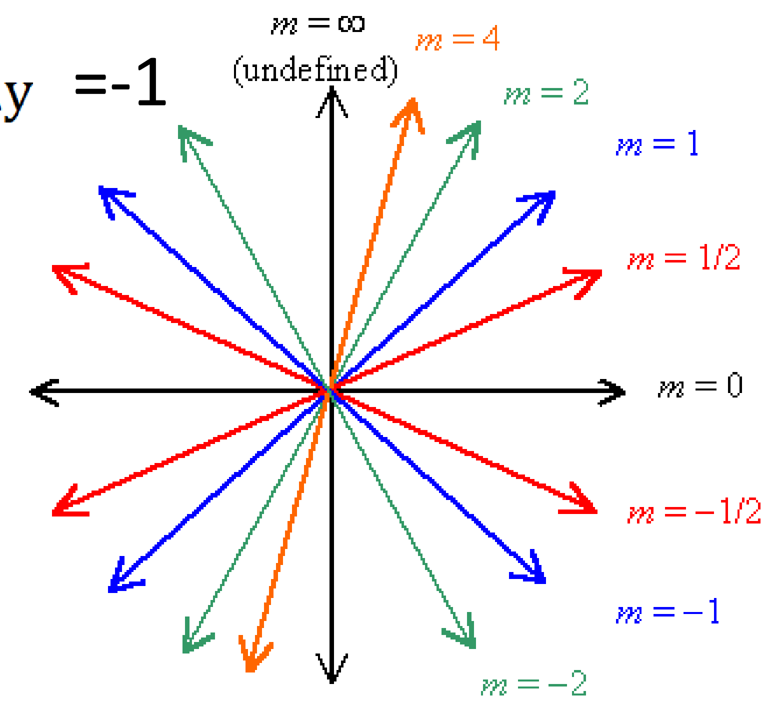
$|\mathrm{m}| = 1, \Delta\mathrm{x} = \Delta\mathrm{y}$ horizontal and vertical deflection voltages are equal

DDA Algorithm
(Digital Differential Analyzer)

- Scan conversion line drawing algorithm based on either Δy or Δx
- sample the line at unit intervals in one coordinate and determine corresponding integer values nearest the line path for the other coordinate
- Consider +ve slope and m<=1; the line is from the left endpoint to the right endpoint
  - Sample at unit x-interval -> Δx =1
  - To compute $y_{k+1} = y_k + m$
  - Initially, k=1; increment 1 until the final endpoint is reached
  - m can be real no; y-value is approximated to nearest integer
- Consider +ve slope and m>1; the line is from the left endpoint to the right endpoint
  - The role is reversed -> $x_{k+1} = x_k + \frac{1}{m}$

DDA Algorithm

- Consider +ve slope and m<=1; the line is from the right endpoint to the left endpoint
  - Sample at unit x-interval -> Δx =-1
  - To compute $y_{k+1} = y_k - m$
- Consider +ve slope and m>1; the line is from the right endpoint to the left endpoint
  - Sample at unit x-interval -> Δy =-1
  - To compute $x_{k+1} = x_k - \frac{1}{m}$



DDA Algorithm: Example

- **Draw a straight line from (2,3) to (8,7) using DDA Algorithm**
- dx=8-2=6;   dy=7-3=4
- m<=1 but positive
- Steps= dx=6; y_{inc}= dy/steps=0.67

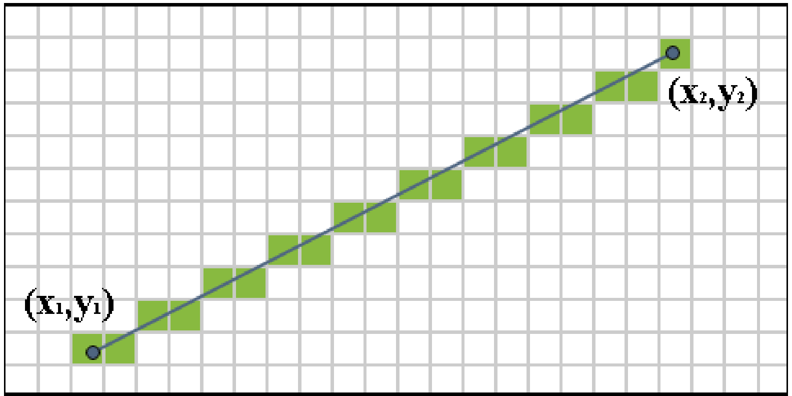
Xold	Yold	Xnew	Ynew	Round(x)	Round(y)
2	3	3	3.67	3	4
3	3.67	4	4.34	4	4
4	4.34	5	5.01	5	5
5	5.01	6	5.68	6	6
6	5.68	7	6.35	7	6
7	6.35	8	7.02	8	7

DDA Algorithm: Pros and Cons

- Faster method for calculating pixel positions than the direct use slope-intercept equation
- Eliminates the multiplication by making use of raster characteristics, so that appropriate increments are applied in the x or y direction to step to pixel positions along the line path
- The accumulation of round-off error in successive additions of the floating-point increment
  - the calculated pixel positions to drift away from the true line path for long line segments
- The rounding operations and floating-point arithmetic are *time-consuming*
- **Improvement:** *by separating the increments m and 1/ m into integer and fractional parts* so that all calculation are reduced to integer operations

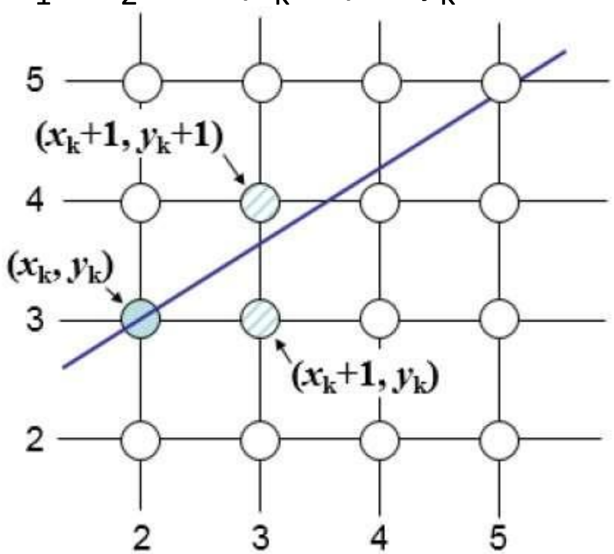
Bresenham's Algorithm

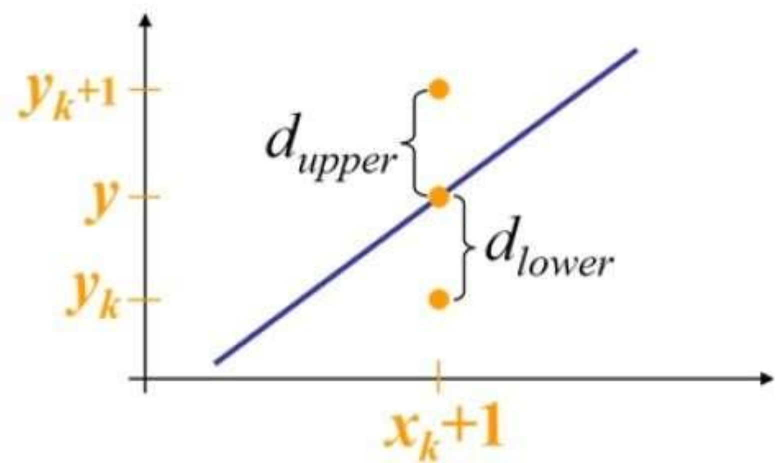
- Developed by J. E. Bresenham
- An accurate and efficient raster line-generating algorithm
- Scan converts lines using only incremental integer calculations
- m<=1 and positive
- sampling at unit x intervals
- Starting from the left endpoint (x_0, y_0) of a given line, we step to each successive column (x position) and plot the pixel whose scan-line y value is closest to the line path



Bresenham's Algorithm

- The pixel (x_k,y_k) is displayed
  - Next pixel to determine -> (x_k+1,y_k) or (x_k+1,y_k+1)
- At sampling position x_k+1, we label vertical pixel separations from the mathematical line path as ***d_1 and d_2***
- The y-coordinate on the mathematical line at pixel column position x_k+1 is calculated as: $y=m(x_k+1)+b$
  - $d_1=y-y_k=m(x_k+1)+b-y_k$
  - $d_2=(y_k+1)-y=(y_k+1)-m(x_k+1)-b$
  - $d_1 - d_2=2m(x_k+1) -2y_k +2b-1$





Bresenham's Algorithm

- Putting the value of m we get the following decision parameter

$$p_k = \Delta x(d_1 - d_2)$$
$$= 2\Delta y \cdot x_k - 2\Delta x \cdot y_k + c$$

- Constant c is $2\Delta y + \Delta x(2b - 1)$
- Δx is positive -> p_k and d_1-d_2 are of same sign
- y_k is closer to the line path ($d_1<d_2$) is $p_k<0$, plot lower pixel
  - Otherwise plot upper pixel
- Use incremental integer calculations
- The decision parameter in step k+1, $p_{k+1} = 2\Delta y \cdot x_{k+1} - 2\Delta x \cdot y_{k+1} + c$
- As $x_{k+1} = x_k + 1$ $\quad p_{k+1} - p_k = 2\Delta y(x_{k+1} - x_k) - 2\Delta x(y_{k+1} - y_k)$
- y_{k+1}-y_k=0 depending on the sign of p_k
- Initial Parameter $p_0 = 2\Delta y - \Delta x$
- The following constants are calculated once for each line to be scan converted $2\Delta y$ and $2\Delta y - 2\Delta x$

Bresenham's Line-Drawing Algorithm for $|m| < 1$

1. Input the two line endpoints and store the left endpoint in (x_0, y_0).
2. Load (x_0, y_0) into the frame buffer; that is, plot the first point.
3. Calculate constants Δx, Δy, $2\Delta y$, and $2\Delta y - 2\Delta x$, and obtain the starting value for the decision parameter as

$$p_0 = 2\Delta y - \Delta x$$

4. At each x_k along the line, starting at $k = 0$, perform the following test: If $p_k < 0$, the next point to plot is $(x_k + 1, y_k)$ and

$$p_{k+1} = p_k + 2\Delta y$$

   Otherwise, the next point to plot is $(x_k + 1, y_k + 1)$ and

$$p_{k+1} = p_k + 2\Delta y - 2\Delta x$$

5. Repeat step 4 Δx times.

Bresenham's Algorithm: Example

Suppose we want to draw a line starting at pixel (2,3) and ending at pixel (12,8).

dx = 12 – 2 = 10
dy = 8 – 3 = 5
p0 = 2dy – dx = 0

2dy = 10
2dy – 2dx = -10

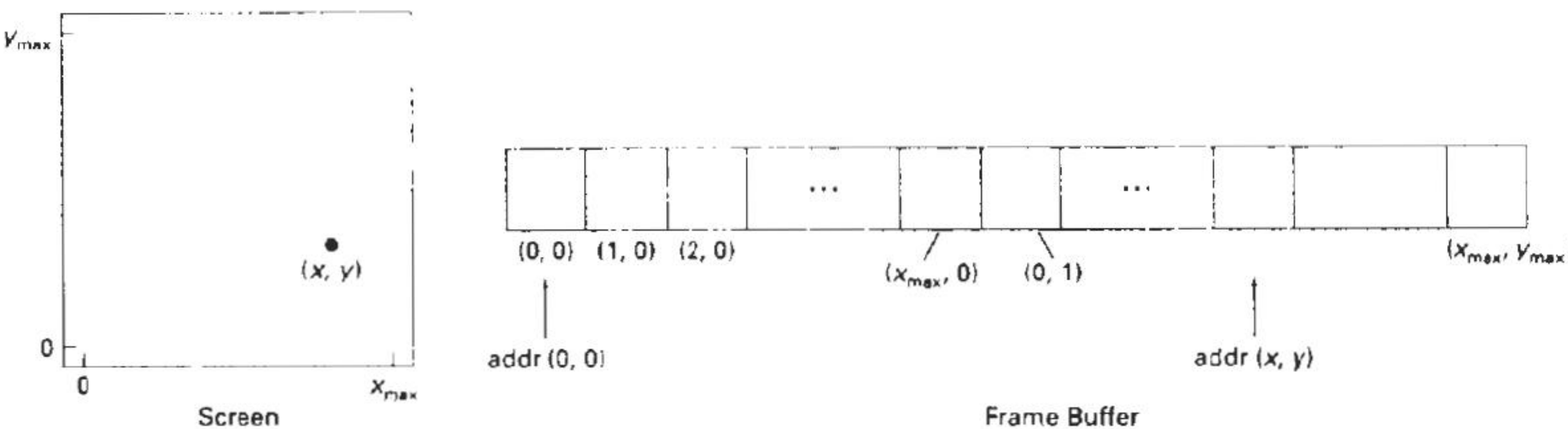
t	p	P(x)	P(y)
0	0	2	3
1	-10	3	4
2	0	4	4
3	-10	5	5
4	0	6	5
5	-10	7	6
6	0	8	6
7	-10	9	7
8	0	10	7
9	-10	11	8
10	0	12	8

Loading Frame Buffer

$$\text{addr}(x, y) = \text{addr}(0, 0) + y(x_{max} + 1) + x$$

$$\text{addr}(x + 1, y) = \text{addr}(x, y) + 1$$

$$\text{addr}(x + 1, y + 1) = \text{addr}(x, y) + x_{max} + 2$$

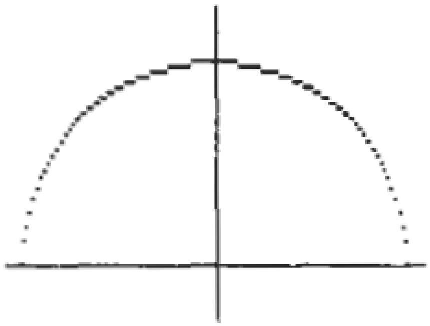


Properties of a Circle

- A circle is defined as the set of points that are all at a given distance r from a center position (x_c, y_c)
- This distance relationship is expressed by the Pythagorean theorem in Cartesian coordinates as $(x - x_c)^2 + (y - y_c)^2 = r^2$
- To calculate the position of points on a circle circumference by stepping along the x axis in unit steps from x_c- r to x_c+ r and calculating the corresponding y values at each position as

$$y = y_c \pm \sqrt{r^2 - (x_c - x)^2}$$

- Involves considerable computation at each step
- Spacing between plotted pixel positions is not uniform
  - By interchanging *x and y* whenever the absolute value of the slope of the circle is greater than 1
  - increases the computation and processing



Properties of a Circle

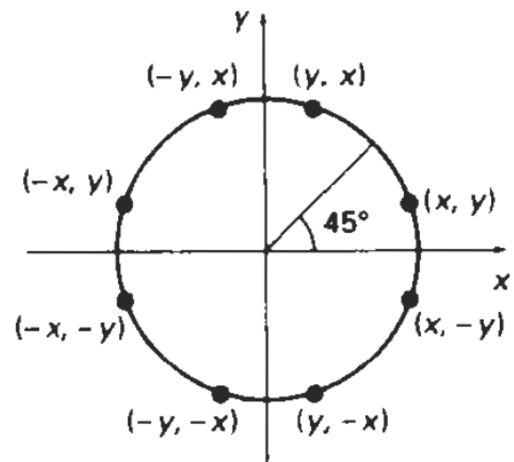
- To eliminate the unequal spacing use polar coordinates : r and Θ

$$x = x_c + r\cos\theta$$

$$y = y_c + r\sin\theta$$

  – Using a fixed angular step size, a circle is plotted with equally spaced points along the circumference
  – Step size depends on the application and display device
  – Larger step size gaps are connected by straight line segment
  – If step: 1/r -> more continuous display
- Computation can be reduced by considering the symmetry of circles
- Bresenham's line algorithm for raster displays is adapted

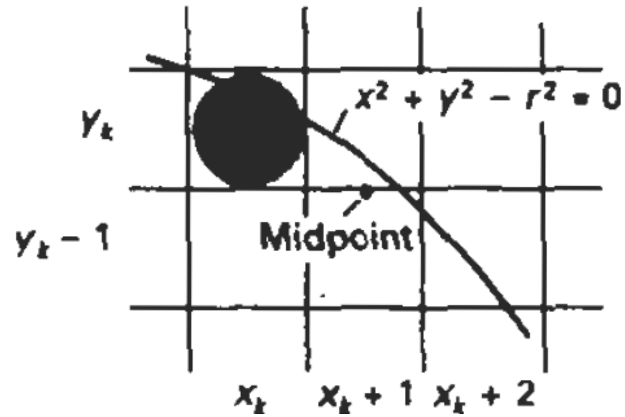
•Direct distance comparison is to test the halfway position between two pixels to determine if this midpoint is inside or outside the circle boundary
•This method is more easily applied to other conics
•Also, the error involved in locating pixel positions along any conic section using the midpoint test is limited to one-half the pixel separation



Midpoint Circle Algorithm

- Circle function: $f_{circle}(x, y) = x^2 + y^2 - r^2$
- Position of a point:

$$f_{circle}(x,y) \begin{cases} < 0, & \text{if } (x,y) \text{ is inside the circle boundary} \\ = 0, & \text{if } (x,y) \text{ is on the circle boundary} \\ > 0, & \text{if } (x,y) \text{ is outside the circle boundary} \end{cases}$$



- Current position: x_k, y_k
- Next point closer to circle: x_k+1, y_k or x_k+1, y_k-1
- The decision parameter is the circle function

$$p_k = f_{circle}\left(x_k + 1, y_k - \frac{1}{2}\right)$$
$$= (x_k + 1)^2 + \left(y_k - \frac{1}{2}\right)^2 - r^2$$

- $p_k < 0$ -> midpoint is inside the circle
  - y_k ; closer to the circle boundary
  - y_k-1; outside the circle boundary

Midpoint Circle Algorithm

- Successive decision parameters are obtained using incremental calculations

$$p_{k+1} = f_{\text{circle}}\left(x_{k+1} + 1, y_{k+1} - \frac{1}{2}\right)$$

$$= [(x_k + 1) + 1]^2 + \left(y_{k+1} - \frac{1}{2}\right)^2 - r^2$$

- Increments: w $p_{k+1} = p_k + 2(x_k + 1) + (y_{k+1}^2 - y_k^2) - (y_{k+1} - y_k) + 1$

$$2x_{k+1} + 1 \qquad 2x_{k+1} + 1 - 2y_{k+1}$$

$$2x_{k+1} = 2x_k + 2$$

$$2y_{k+1} = 2y_k - 2$$

- Start position: (0,r) -> $2x_{k+1}=0$; $2y_{k+1}=2r$
- Rounding p_0 to an integer: $p_0 = 1-r$

$$p_0 = f_{\text{circle}}\left(1, r - \frac{1}{2}\right)$$

$$= 1 + \left(r - \frac{1}{2}\right)^2 - r^2$$

$$p_0 = \frac{5}{4} - r$$

Midpoint Circle Algorithm

1. Input radius r and circle center (x_c, y_c), and obtain the first point on the circumference of a circle centered on the origin as

$$(x_0, y_0) = (0, r)$$

2. Calculate the initial value of the decision parameter as

$$p_0 = \frac{5}{4} - r$$

3. At each x_k position, starting at $k = 0$, perform the following test: If $p_k < 0$, the next point along the circle centered on $(0, 0)$ is (x_{k+1}, y_k) and

$$p_{k+1} = p_k + 2x_{k+1} + 1$$

Otherwise, the next point along the circle is $(x_k + 1, y_k - 1)$ and

$$p_{k+1} = p_k + 2x_{k+1} + 1 - 2y_{k+1}$$

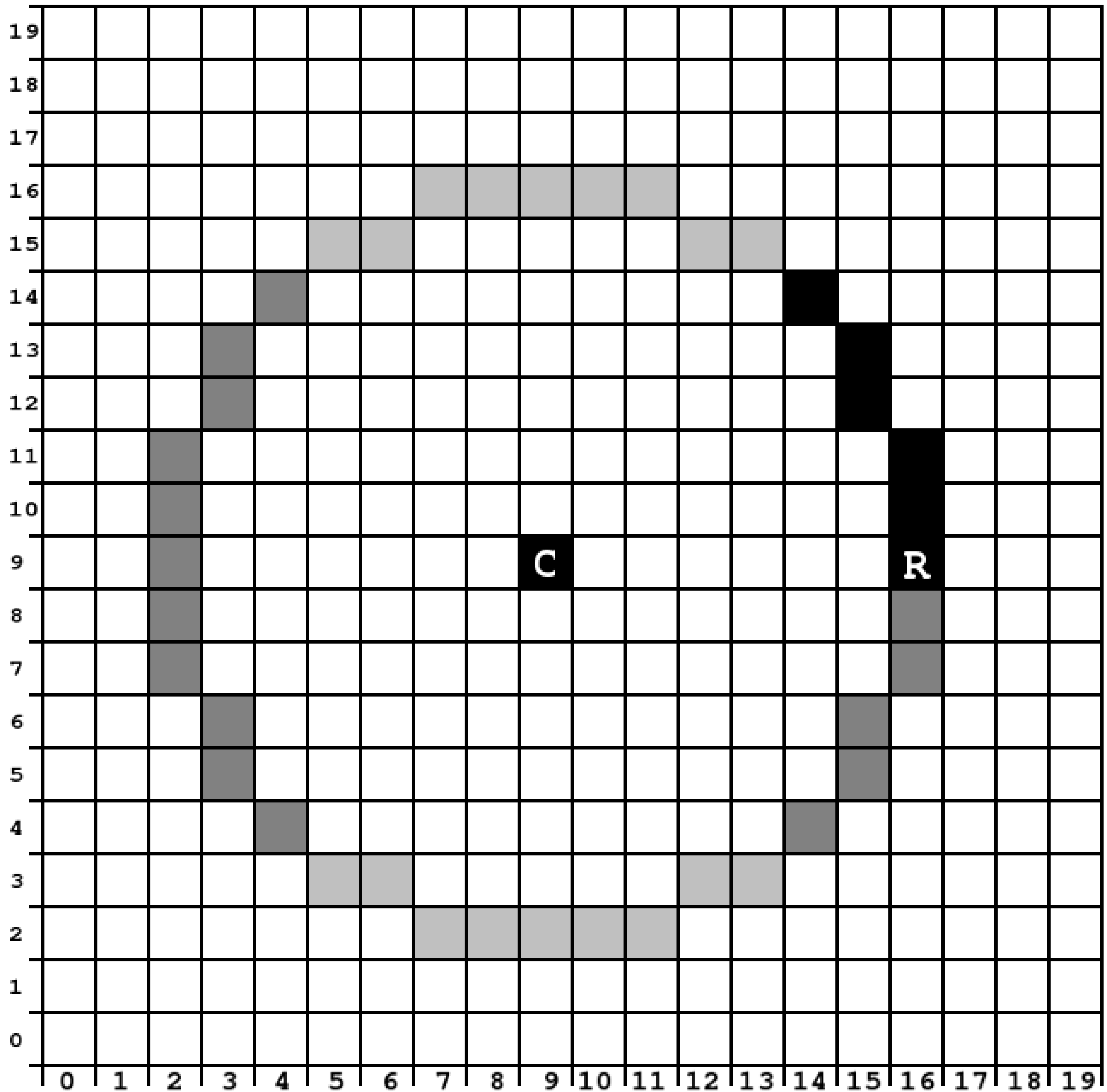
where $2x_{k+1} = 2x_k + 2$ and $2y_{k+1} = 2y_k - 2$.

4. Determine symmetry points in the other seven octants.
5. Move each calculated pixel position (x, y) onto the circular path centered on (x_c, y_c) and plot the coordinate values:

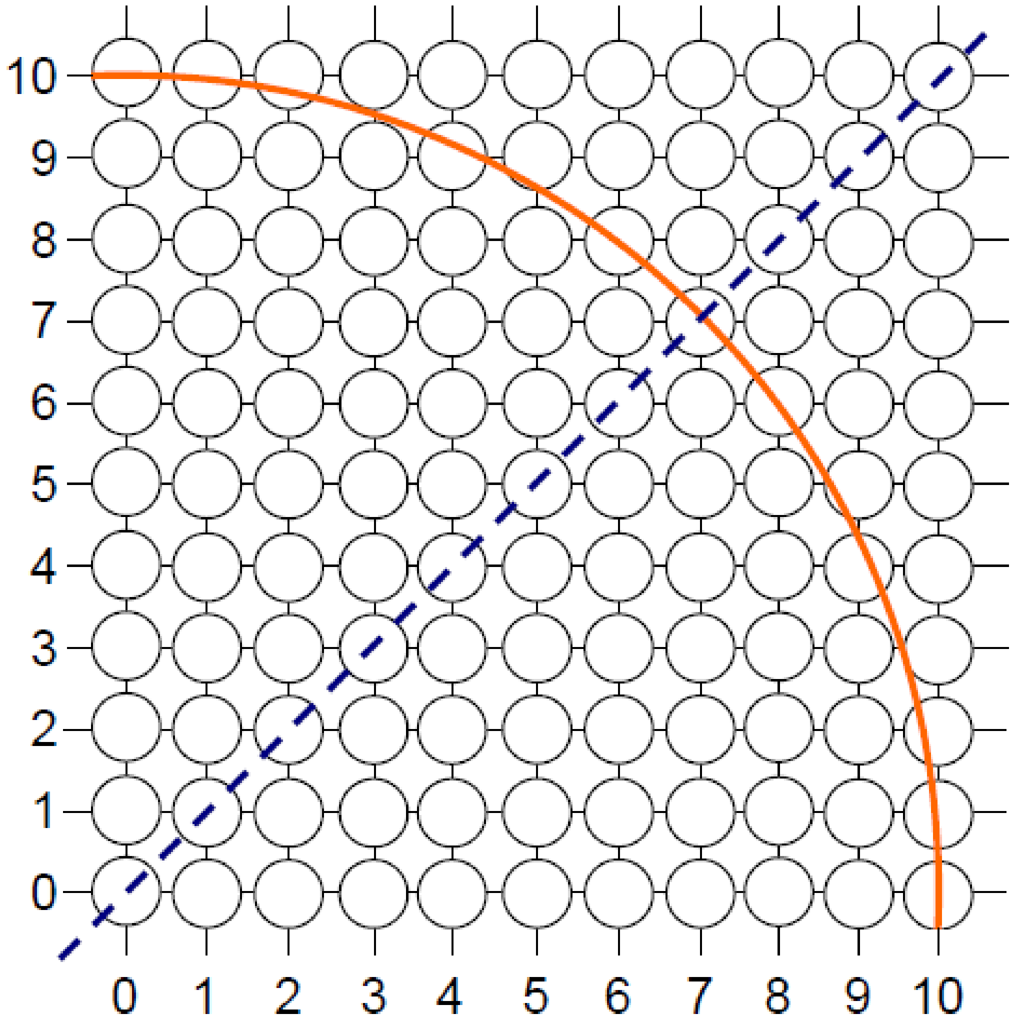
$$x = x + x_c, \qquad y = y + y_c$$

6. Repeat steps 3 through 5 until $x \geq y$.

Midpoint Circle Algorithm



Midpoint Circle Algorithm



k	p_k	(x_{k+1},y_{k+1})	$2x_{k+1}$	$2y_{k+1}$
0				
1				
2				
3				
4				
5				
6				

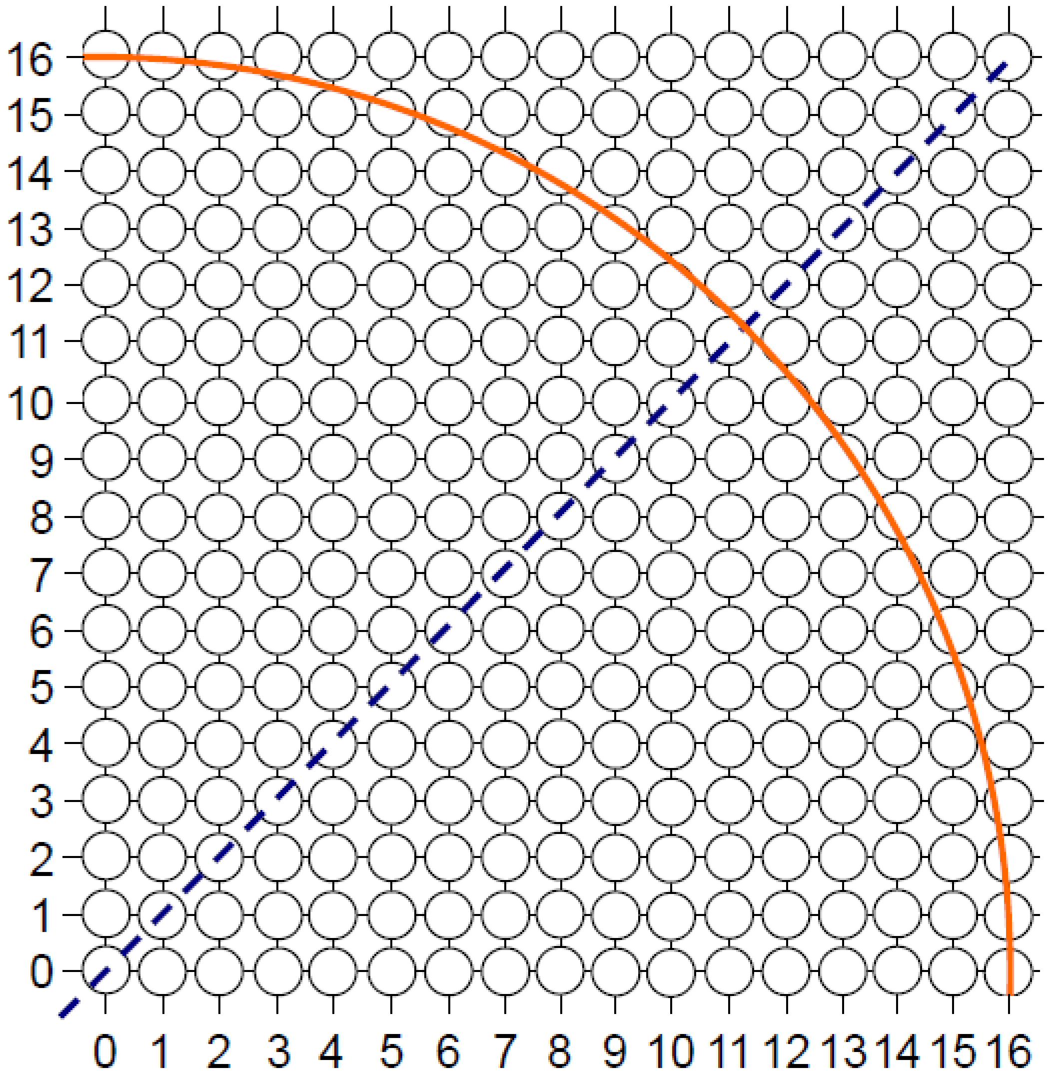
Midpoint Circle Algorithm

To see the mid-point circle algorithm in action lets use it to draw a circle centred at (0,0) with radius 10

k	p_k	(x_{k+1}, y_{k+1})	$2x_{k+1}$	$2y_{k+1}$
0	−9	(1, 10)	2	20
1	−6	(2, 10)	4	20
2	−1	(3, 10)	6	20
3	6	(4, 9)	8	18
4	−3	(5, 9)	10	18
5	8	(6, 8)	12	16
6	5	(7, 7)	14	14

Midpoint Circle Algorithm

To see the mid-point circle algorithm in action lets use it to draw a circle centred at (0,0) with radius 16



k	p_k	(x_{k+1}, y_{k+1})	$2x_{k+1}$	$2y_{k+1}$
0				
1				
2				
3				
4				
5				
6				
7				
8				
9				
10				
11				
12				

Midpoint Circle Algorithm

The key insights in the mid-point circle algorithm are:

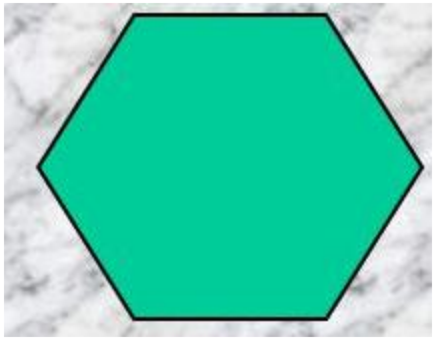
- Eight-way symmetry can hugely reduce the work in drawing a circle
- Moving in unit steps along the x axis at each point along the circle's edge we need to choose between two possible y coordinates

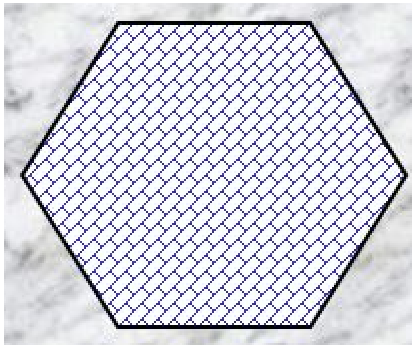
Filled area Primitives

- Solid-color or patterned polygon area
- Polygons are easier to process since they have linear boundaries
- There are two basic approaches for area filling
- To determine the overlap intervals for scan lines that cross the area
- To start from a given interior position and paint outward from this point until we encounter the specified boundary conditions
- The scan-line approach is typically used in general graphics packages to fill polygons circles, ellipses, and other simple curves
- All methods starting from an interior point are useful with more complex boundaries and in interactive painting systems

Filled area Primitives

- Solid Fill
- Pattern Fill





- Polygon of n-vertices, ordered list of edges

- To learn:

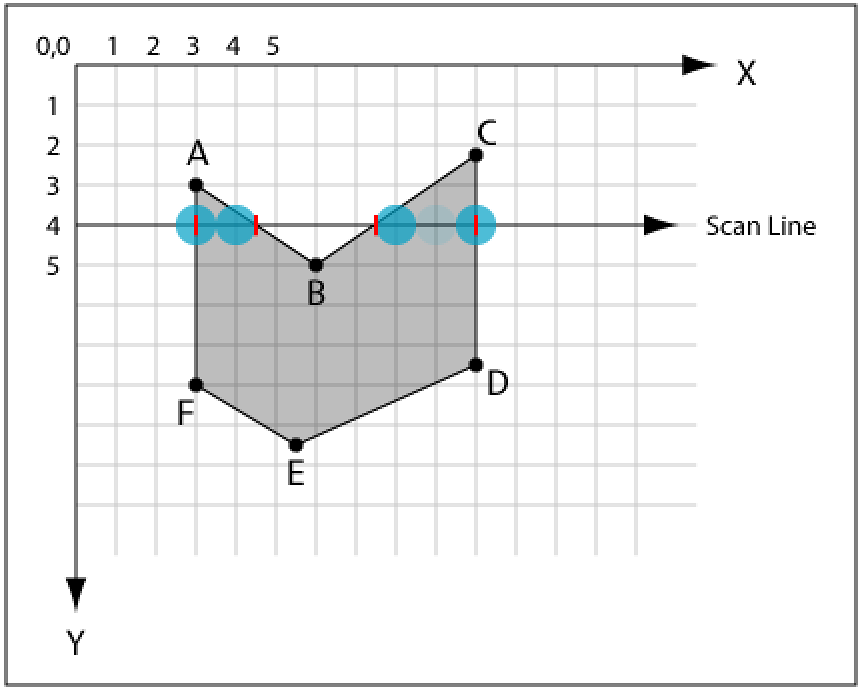
1. Scan-Line Fill Algorithm
2. Flood-Fill Algorithm
3. Boundary-Fill Algorithm

Scan Line Fill Algorithms

- Solid filling of polygonal areas
- For each scan line crossing a polygon, the area-fill algorithm locates the intersection points of the scan line with the polygon edges.
- These intersection points are then sorted from left to right, and the corresponding frame-buffer positions between each intersection

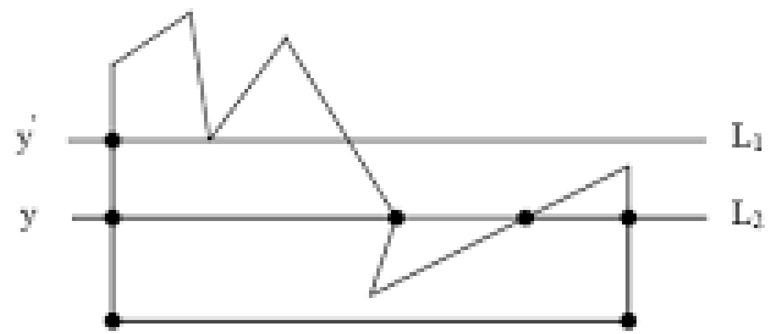
pair are set to the specified

fill color



Scan Line Fill Algorithms

- Scan-line intersection with polygon vertices needs special handling
  - Intersects two polygon edges
- The corresponding edges either same side of the scan line or opposite side of it
- Identify these vertices by



traversing the polygon boundary in clockwise or anti-clockwise manner

-- observing the relative changes in y-direction

-- y-value increases or decreases monotonically -> count single

-- otherwise, count twice

Scan Line Fill Algorithms

Adjusting endpoint values for a polygon, as we process edges in order around the polygon perimeter. The edge currently being processed is indicated as a solid like. In (a), the y coordinate of the upper endpoint of the current edge id decreased by 1. In (b), the y coordinate of the upper end point of the next edge is decreased by 1

Scan Line Fill Algorithms

- Graphics algorithm takes advantage of coherence of a scene
- Properties of one part of a scene is somewhat related to other part of a scene
- Involves in incremental calculations, applied along a scan line or successive scan lines
- To determine edge intersection, incremental coordinate calculations
  - Slope of the line remains constant from one scan line to the other
  - $m=(y_{k+1}-y_k)/(x_{k+1}-x_k)$; changes in y-coordinates: $y_{k+1}-y_k=1$
  - x-coordinate can also be determined: $x_{k+1}-x_k=1/m$

Scan Line Fill Algorithms

- The x_k value of k^{th} scan line of slope m is $x_k = x_0 + \frac{k}{m}$
- The increment in x-value is 1/m $m = \frac{\Delta y}{\Delta x}$ $x_{k+1} = x_k + \frac{\Delta x}{\Delta y}$

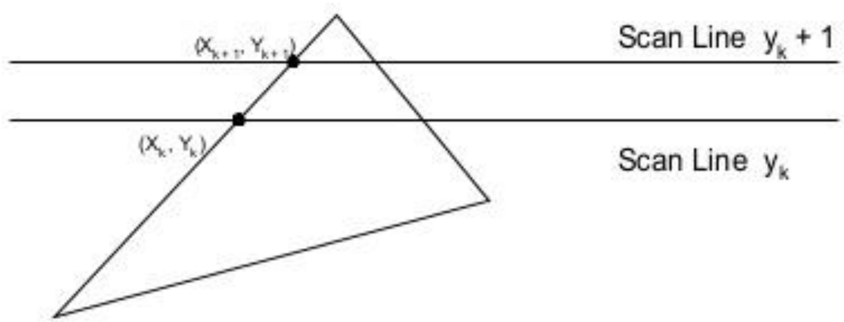
The scan conversion algorithm works as follows

i. Intersect each scanline with all edges
ii. Sort intersections in x
iii. Calculate parity of intersections to determine in/out
iv. Fill the "in" pixels

Special cases to be handled:

i. Horizontal edges should be excluded
ii. Vertices lying on scanlines handled by shortening of edges,

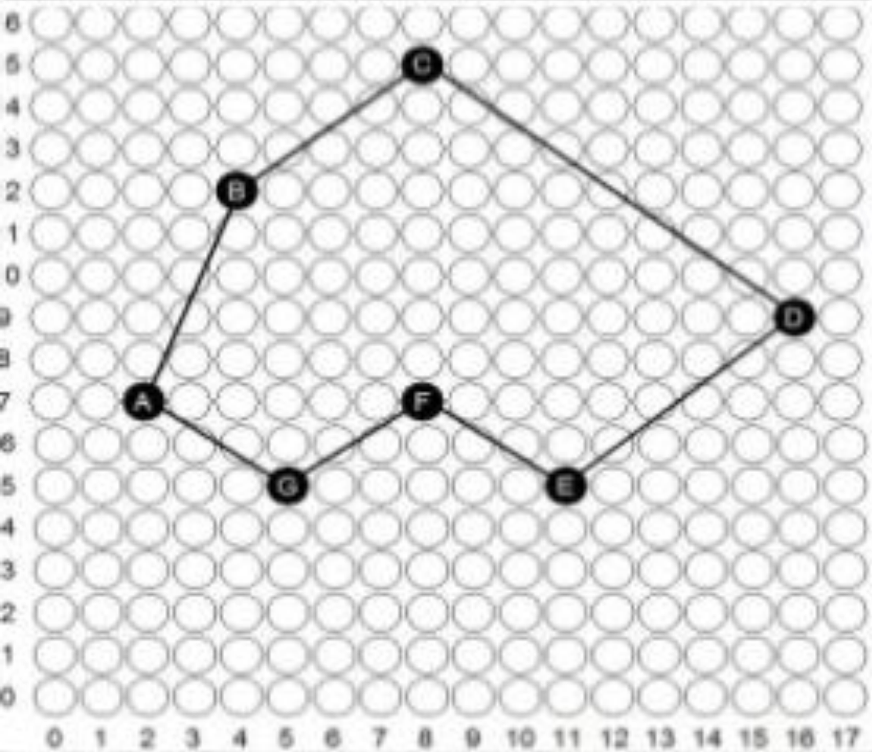
- Coherence between scanlines tells us that
  - Edges that intersect scanline *y* are likely to intersect *y + 1*
  - *X* changes predictably from scanline *y* to *y + 1 (Incremental Calculation Possible)*



Scan Line Fill Algorithms

(Example)

Edge Table						
#	Edge		1/m	y_{min}	x	y_{max}
0	A (2, 7)	B (4, 12)	2/5	7	2	12
1	B (4, 12)	C (8,15)	4/3	12	4	15
2	C (8,15)	D (16, 9)	– 8/6	9	16	15
3	D (16, 9)	E (11, 5)	5/4	5	11	9
4	E (11, 5)	F (8, 7)	– 3/2	5	11	7
5	F (8, 7)	G (5, 5)	3/2	5	5	7
6	G (5, 5)	A (2, 7)	– 3/2	5	5	7



Edge number 0

#	Edge		1/m	y_{min}	x	y_{max}
0	A (2, 7)	B' (4, 11)	2/5 = 0.4	7	2	11

Scan line	x-intersection
y = 7	2
y = 8	2 + 0.4 = 2.4 ~ 2
y = 9	2.4 + 0.4 = 2.8 ~ 3
y = 10	2.8 + 0.4 = 3.2 ~ 3
y = 11	4

Edge number 1

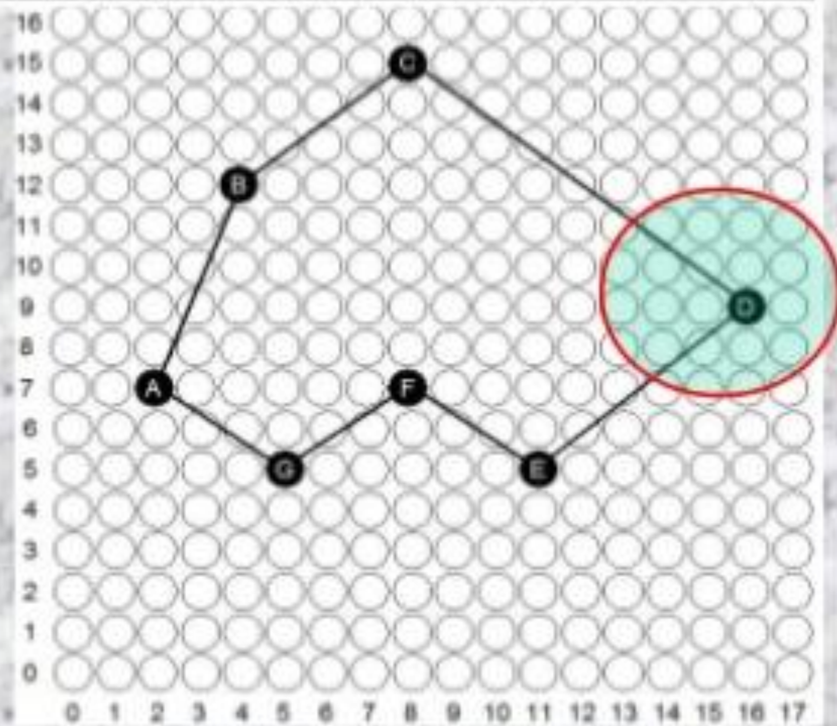
#	Edge		1/m	y_{min}	x	y_{max}
1	B (4, 12)	C (8,15)	4/3 = 1.3	12	4	15

Scan line	x-intersection
y = 12	4
y = 13	4 + 1.3 = 4.3 ~ 4
y = 14	4.3 + 1.3 = 5.6 ~ 6
y = 15	8

Scan Line Fill Algorithms

(Example)

Edge Table						
#	Edge		1/m	y_{min}	x	y_{max}
0	A (2, 7)	B (4, 12)	2/5	7	2	12
1	B (4, 12)	C (8,15)	4/3	12	4	15
2	C (8,15)	D (16, 9)	– 8/6	9	16	15
3	D (16, 9)	E (11, 5)	5/4	5	11	9
4	E (11, 5)	F (8, 7)	– 3/2	5	11	7
5	F (8, 7)	G (5, 5)	3/2	5	5	7
6	G (5, 5)	A (2, 7)	– 3/2	5	5	7



Edge number 2

#	Edge		1/m	y_{min}	x	y_{max}
2	C (8,15)	D (16, 9)	– 8/6 = –1.3	9	16	15

Scan line	x-intersection
y = 9	16
y = 10	16 – 1.3 = 14.7 ~ 15
y = 11	14.7 – 1.3 = 13.4 ~ 13
y = 12	13.4 – 1.3 = 12.1 ~ 12
y = 13	12.1 – 1.3 = 10.8 ~ 11
y = 14	10.8 – 1.3 = 9.5 ~ 10
y = 15	8

Edge number 3

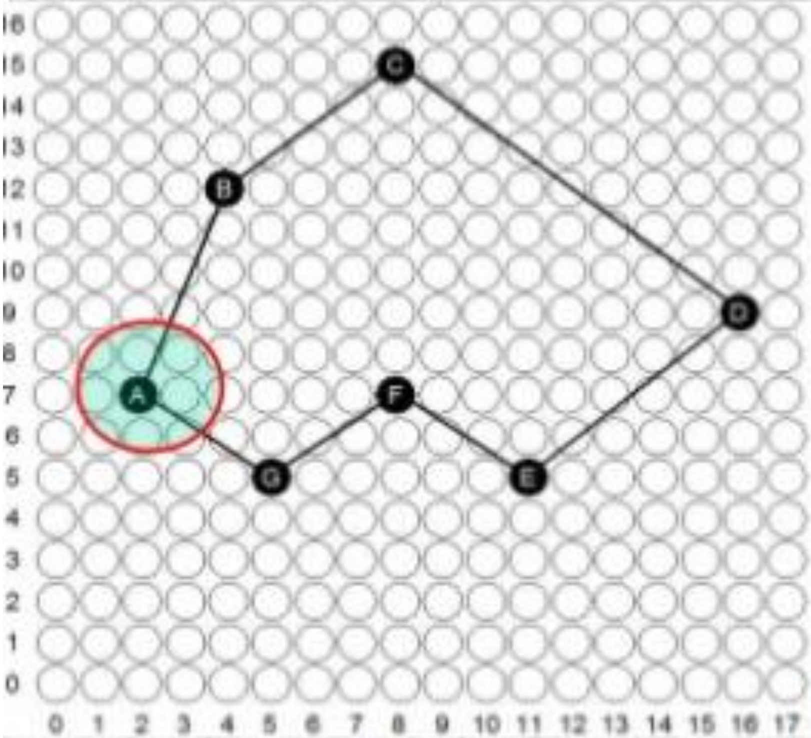
#	Edge		1/m	y_{min}	x	y_{max}
3	D' (15, 8)	E (11, 5)	5/4 = 1.25	5	11	8

Scan line	x-intersection
y = 5	11
y = 6	11 + 1.25 = 12.25 ~ 12
y = 7	12.25 + 1.25 = 13.5 ~ 14
y = 8	15

Scan Line Fill Algorithms

(Example)

Edge Table						
#	Edge		1/m	y_{min}	x	y_{max}
0	A (2, 7)	B (4, 12)	2/5	7	2	12
1	B (4, 12)	C (8, 15)	4/3	12	4	15
2	C (8, 15)	D (16, 9)	– 8/6	9	16	15
3	D (16, 9)	E (11, 5)	5/4	5	11	9
4	E (11, 5)	F (8, 7)	– 3/2	5	11	7
5	F (8, 7)	G (5, 5)	3/2	5	5	7
6	G (5, 5)	A (2, 7)	– 3/2	5	5	7



Edge number 4

#	Edge		1/m	y_{min}	x	y_{max}
4	E (11, 5)	F (8, 7)	– 3/2 = –1.5	5	11	7

Scan line	x-intersection
y = 5	11
y = 6	11 – 1.5 = 9.5 ~ 10
y = 7	8

Edge number 5

#	Edge		1/m	y_{min}	x	y_{max}
5	F (8, 7)	G (5, 5)	3/2 = 1.5	5	5	7

Scan line	x-intersection
y = 5	5
y = 6	5+ 1.5 = 6.5 ~ 7
y = 7	8

Edge number 6

#	Edge		1/m	y_{min}	x	y_{max}
6	G (5, 5)	A' (4, 6)	– 3/2 = –1.5	5	5	6

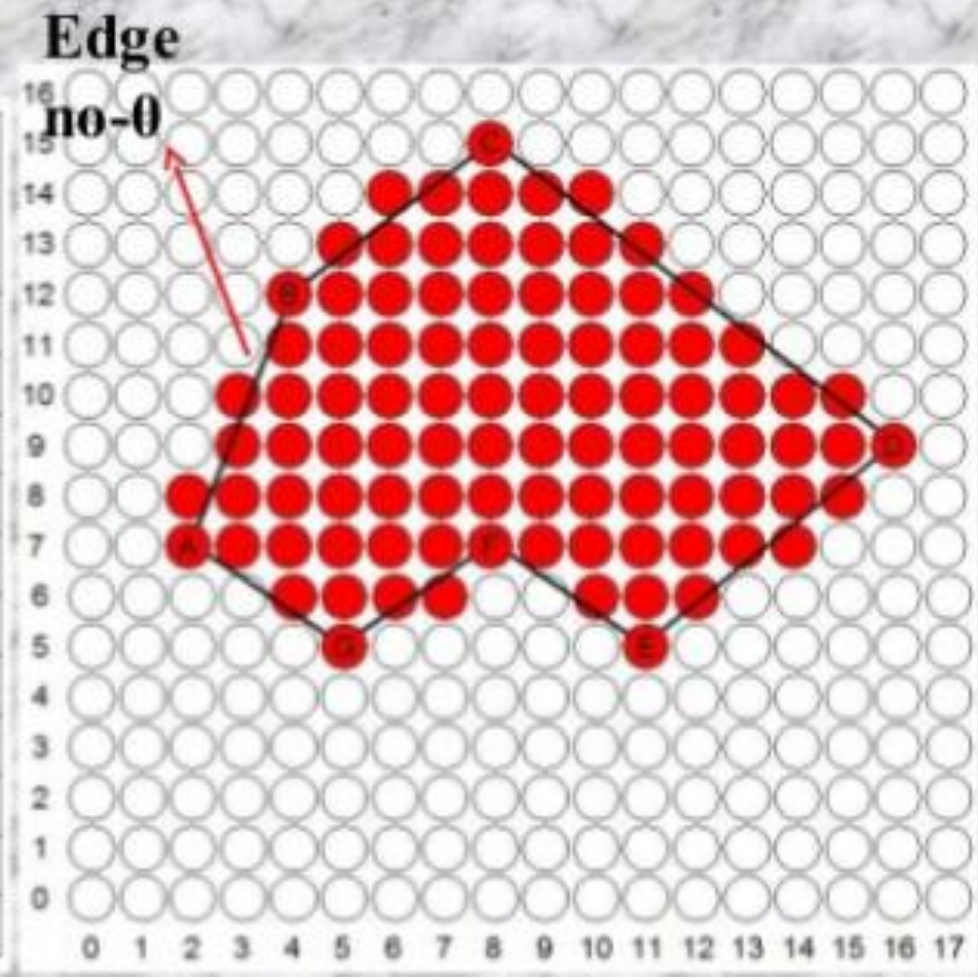
Scan line	x-intersection
y = 5	5
y = 6	4

Correction: A'(3,6)

Scan Line Fill Algorithms

(Example)

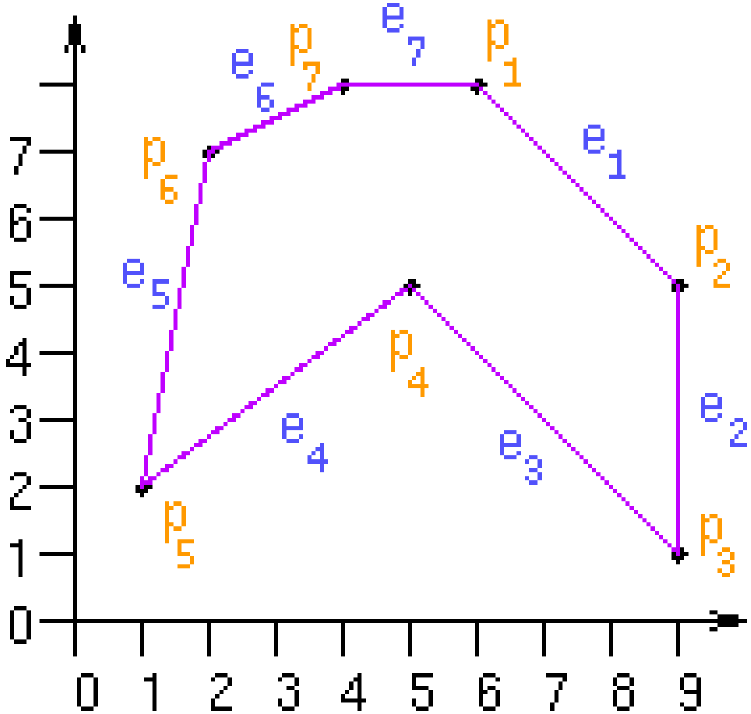
Scan line	x-intersections Edge# 0	1	2	3	4	5	6	x-intersections pair Ascending order
5				11	11	5	5	(5, 5), (11, 11)
6				12	10	7	4	(4, 7), (10, 12)
7	2			14	8	8		(2, 8), (8,14)
8	2			15				(2,15)
9	3		16					(3, 16)
10	3		15					(3, 15)
11	4		13					(4, 13)
12		4	12					(4,12)
13		4	11					(4, 11)
14		6	10					(6, 10)
15		8	8					(8, 8)

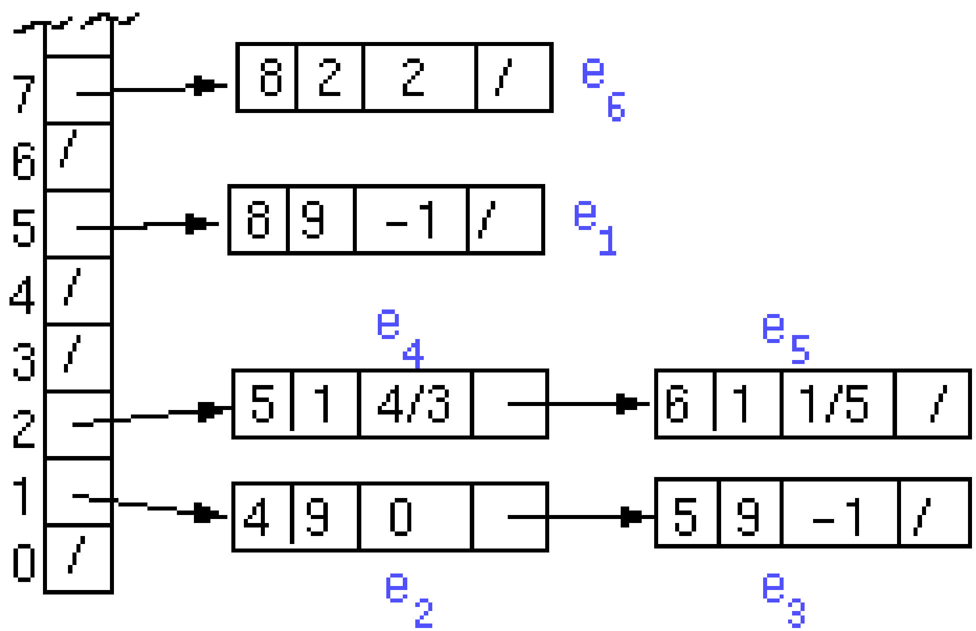


Scan Line Fill Algorithms: AEL

- Process the scan lines from bottom to top to construct Active Edge Table during scan conversion.
- Maintain an active edge list for the current scan-line.
- When the current scan line reaches the lower / upper endpoint of an edge it becomes active.
- When the current scan line moves above the upper / below the lower endpoint, the edge becomes inactive
- Use iterative coherence calculations to obtain edge intersections quickly.
- AEL is a linked list of active edges on the current scanline, y.
  - Each active edge line has the following information
    - y_upper: last scanline to consider
    - x_lower: edge's intersection with current y
    - 1/m: x increment
  - The active edges are kept sorted by x

Scan Line Fill Algorithms: AEL





Scan Line Polygon Fill Algorithm

1. Set y to the smallest y coordinate that has an entry in the ET; *i.e*, y fo the first nonempty bucket.
2. Initialize the AET to be empty.
3. Repeat until the AET and ET are empty:
   1. Move from ET bucket y to the AET those edges whose $y\_min = y$ (entering edges).
   2. Remove from the AET those entries for which $y = y\_max$ (edges not involved in the next scanline), the sort the AET on x (made easier because ET is presorted
   3. Fill in desired pixel values on scanline y by using pairs of x coordinates from AET.
   4. Increment y by 1 (to the coordinate of the next scanline).
   5. For each nonvertical edge remaining in the AET, update x for the new y.

Inside-Outside Test

- To determine whether a point on the scan line lies inside or outside the polygon
- Mostly used to identify a point in the hollow polygon

- Two Methods:

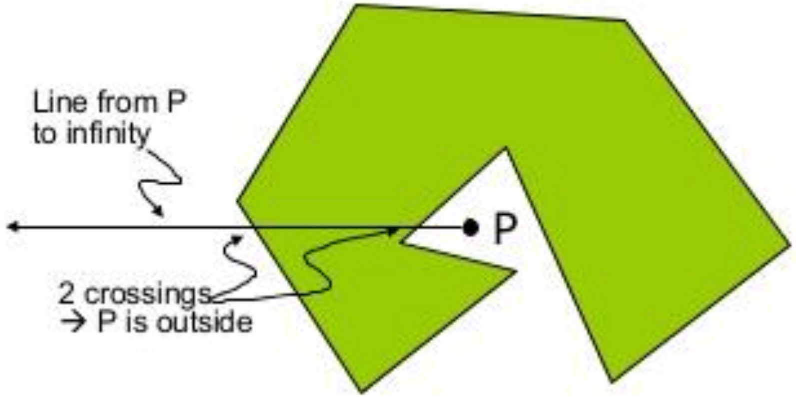
1. Even-Odd Rule, Odd-Even Rule, Odd Parity Rule
2. Nonzero Winding number rule

Inside-Outside Test

Even-Odd Parity Rule

Inside-outside test for a point P:

1. Draw line from P to infinity
   - Any direction
   - Does not go through any vertex
2. Count the number of times the line crosses an edge
   - If the number of crossings is odd, P is inside
   - If the number of crossings is even, P is outside



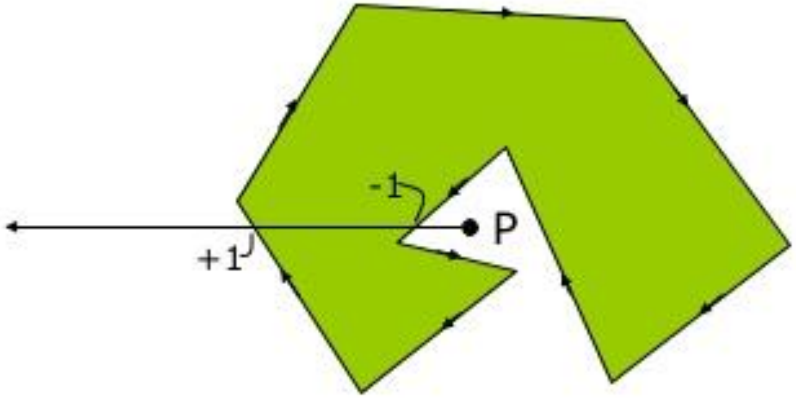
Inside-Outside Test

Non-Zero Winding Number Rule

Inside-outside test:

1. Determine the **winding number** W of P
   a. Initialize W to zero and draw a line from P to infinity
   b. If the line crosses an edge directed from bottom to top, W++
   c. If the line crosses an edge directed from top to bottom, W--

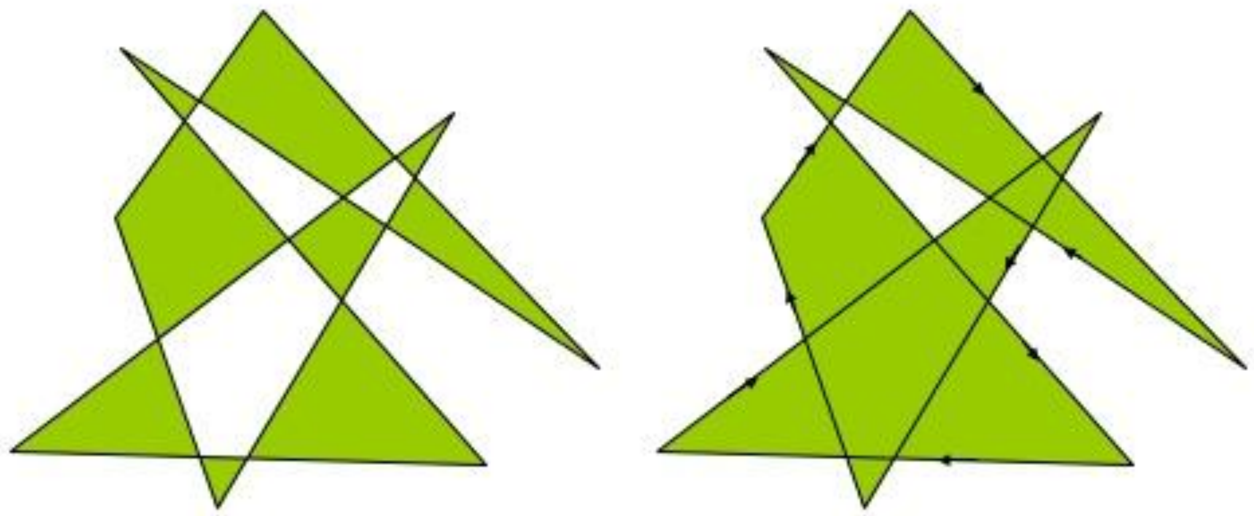
2. If the W = 0, P is outside
3. Otherwise, P is inside



Inside-Outside Test

General polygons

Can be self intersecting
Can have interior holes



Even-odd parity   Non-zero winding

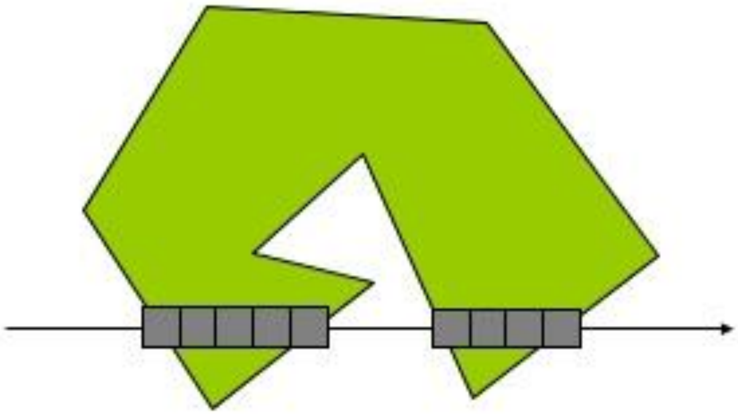
The non-zero winding number rule and the even-odd parity rule can give different results for general polygons

Inside-Outside Test

Raster-Based Filling

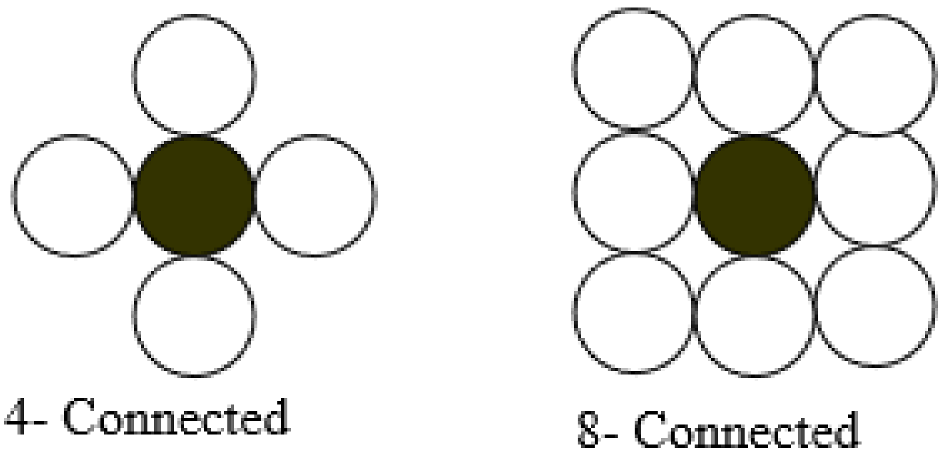
For each scan line

- Determine points where the scan line intersects the polygon
- Set pixels between intersection points (using a fill rule)
  - Even-odd parity rule: set pixels between pairs of intersections
  - Non-zero winding rule: set pixels according to the winding number



Boundary Fill Algorithms

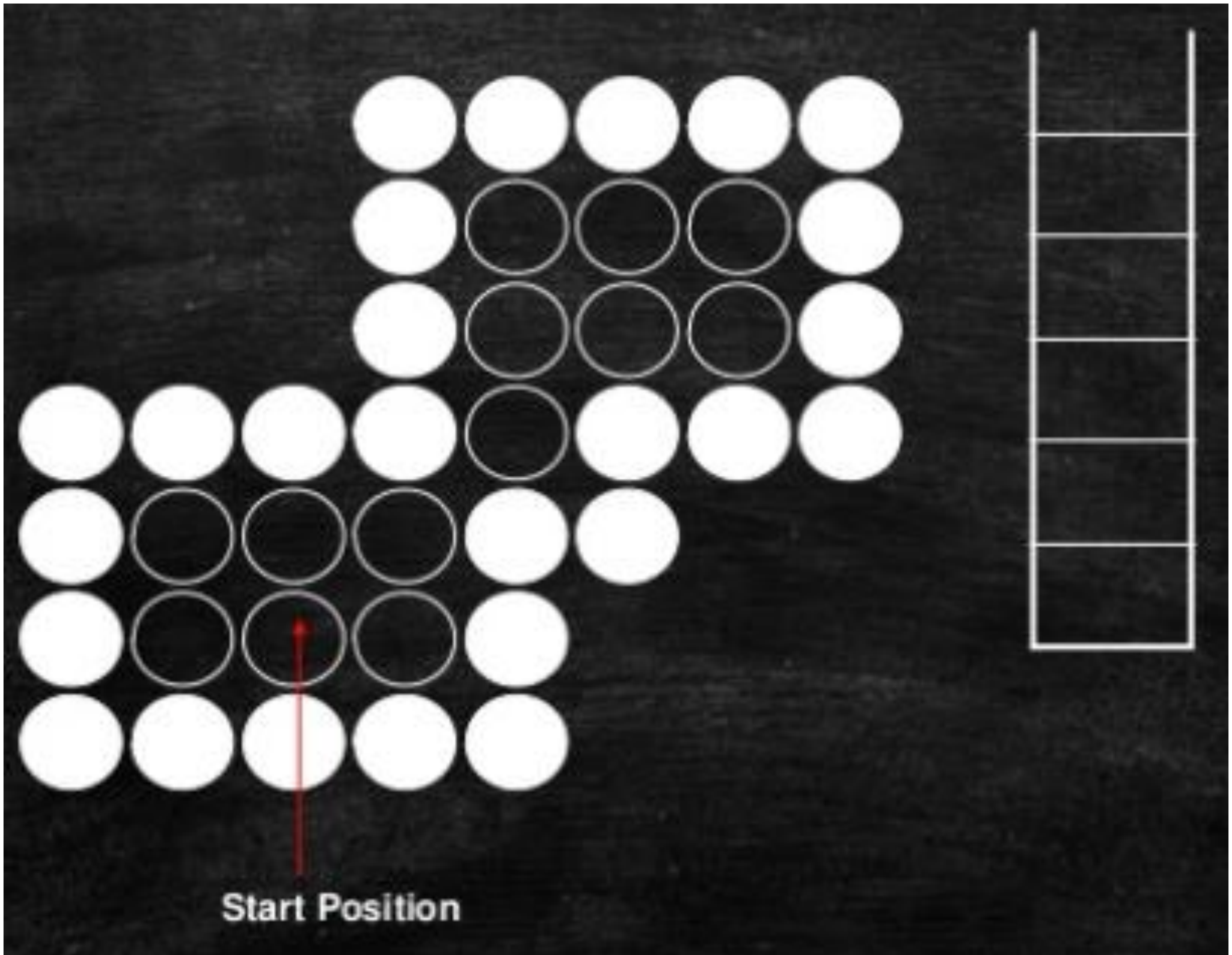
- To start from a interior point and paint the interior outward toward the boundary
- If the boundary is specified by a single color, the fill algorithm processed outward pixel by pixel until the boundary color is encountered
- A boundary fill procedure accepts an interior point (x,y), a fill color, and a boundary color
- Based on connectivity the filling is of two types



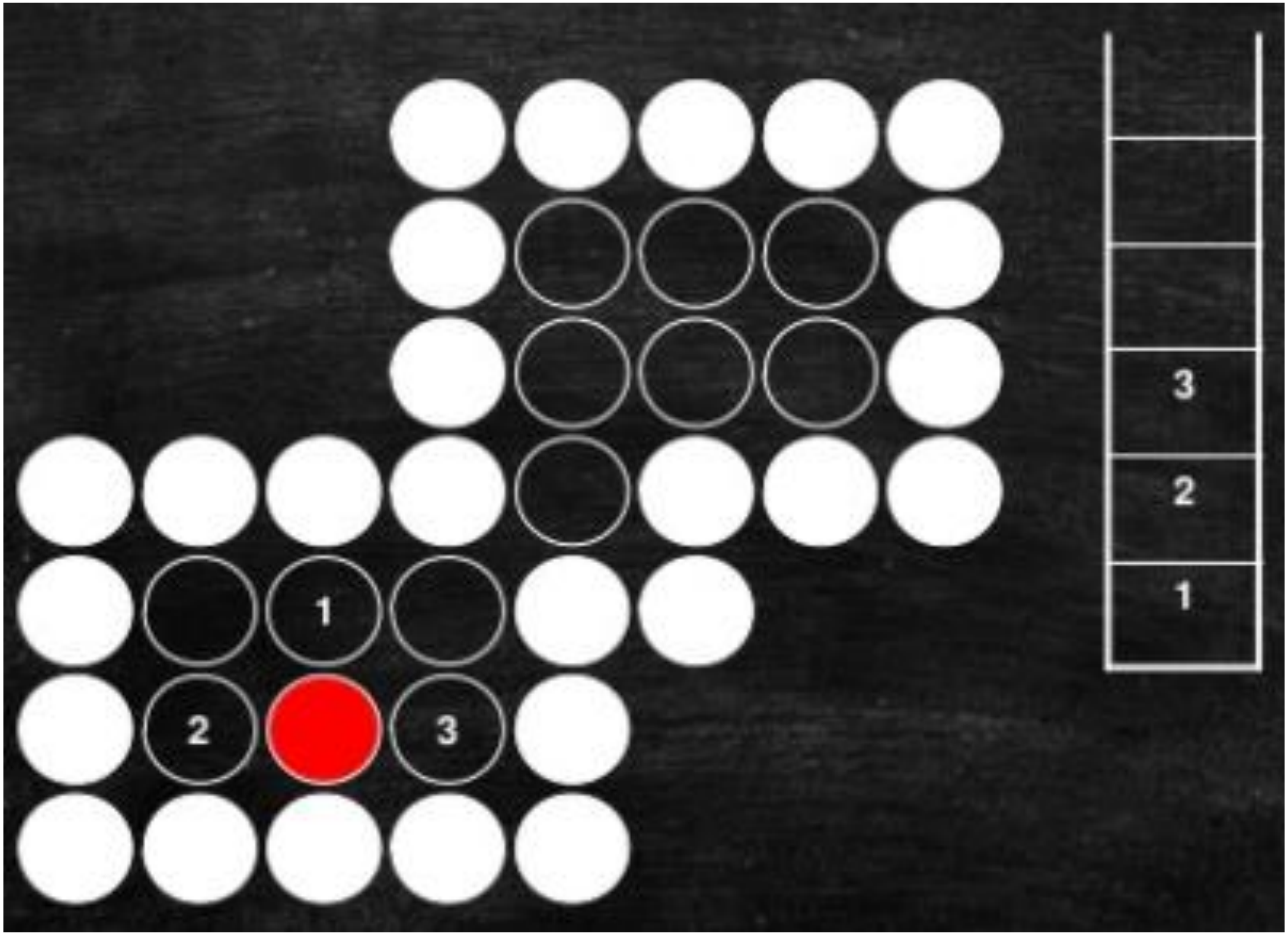
Boundary Fill Algorithms

```
void boundaryFill(int x, int y,
          int fillColor, int borderColor)
{
  getPixel(x, y, color);
  if ((color != borderColor)
          && (color != fillColor)) {
     setPixel(x,y);
     boundaryFill(x+1,y,fillColor,borderColor);
     boundaryFill(x-1,y,fillColor,borderColor);
     boundaryFill(x,y+1,fillColor,borderColor);
     boundaryFill(x,y-1,fillColor,borderColor);
  }
}
```

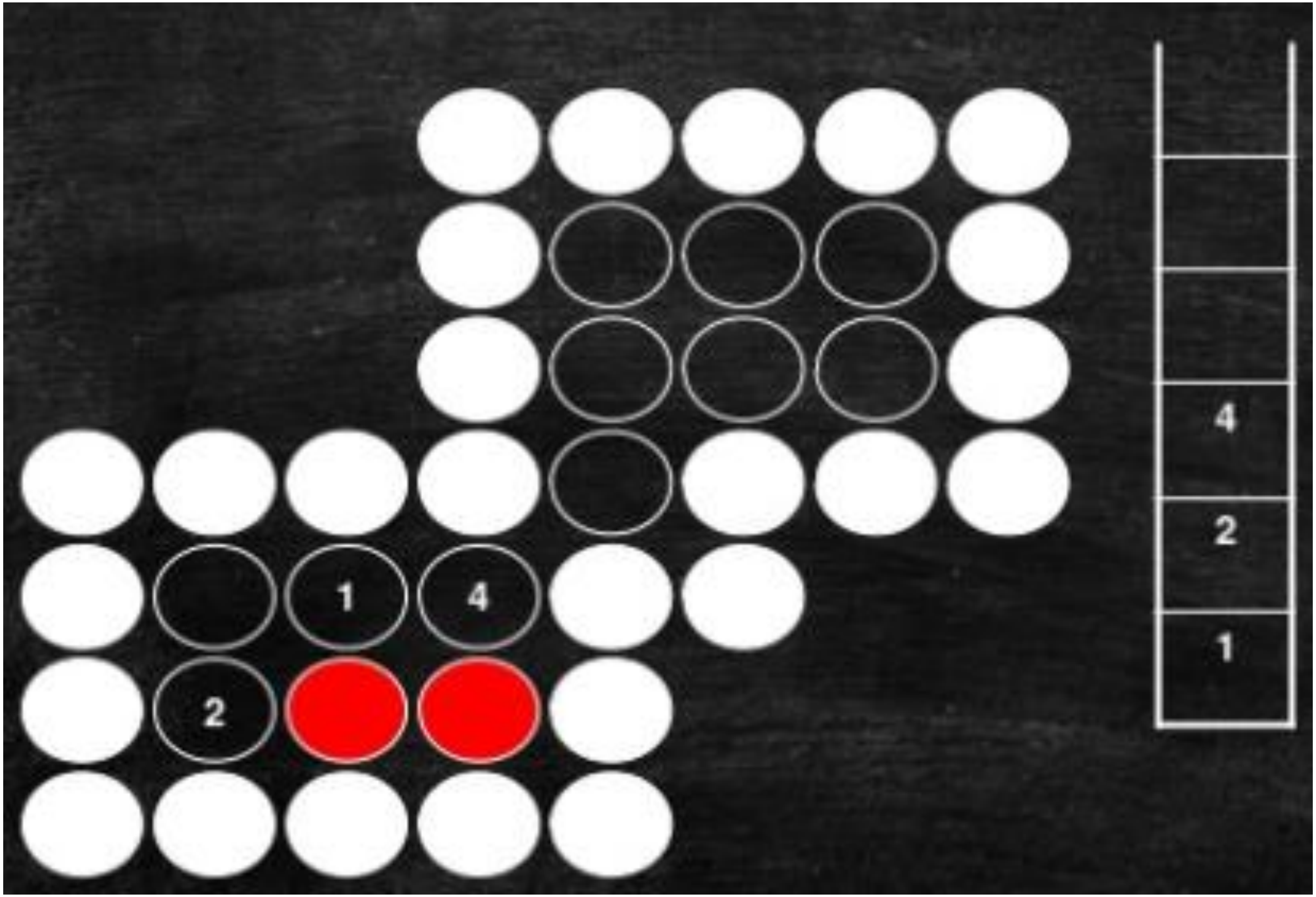
Boundary Fill Algorithms Example: 4-connected



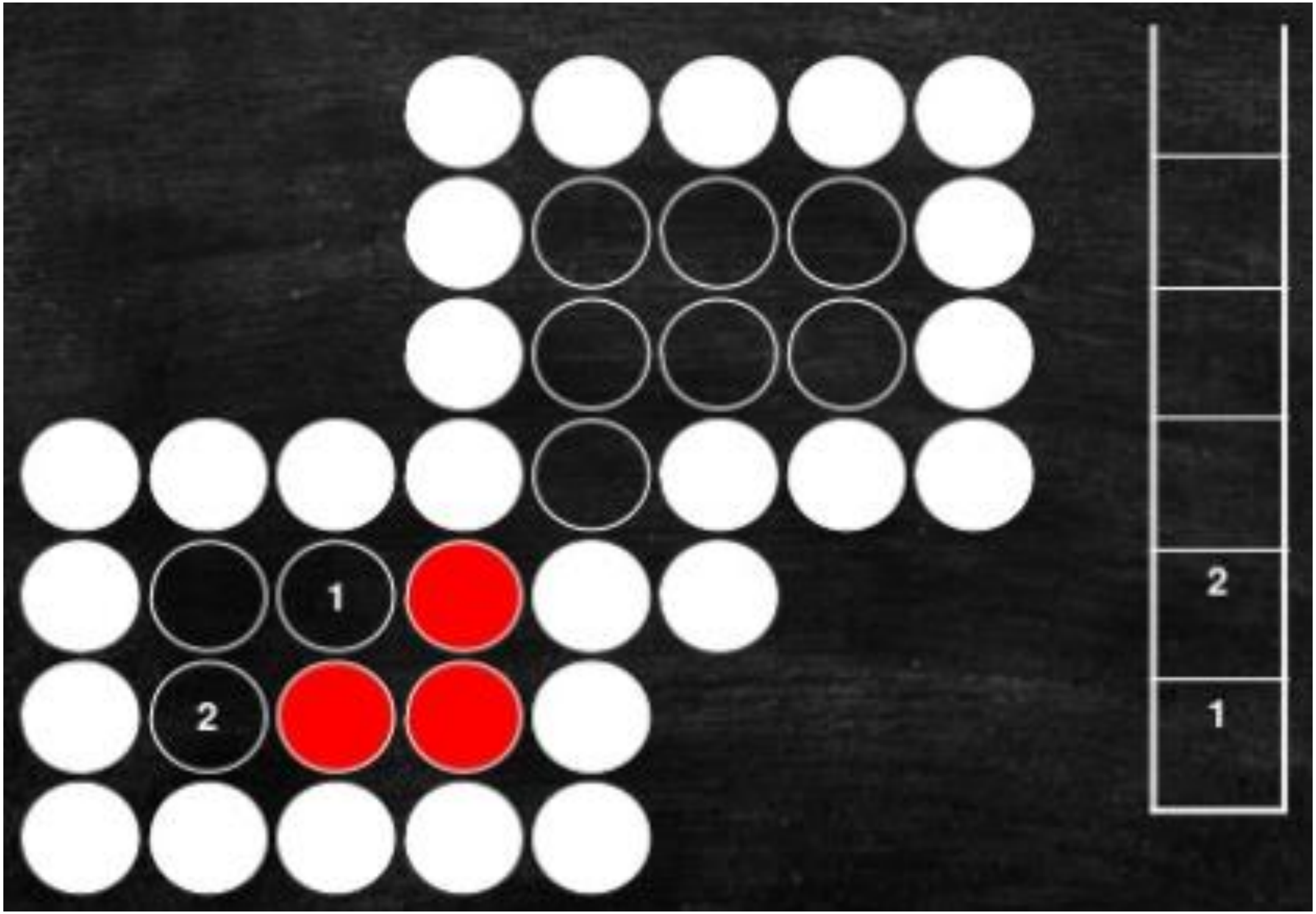
Boundary Fill Algorithms Example: 4-connected



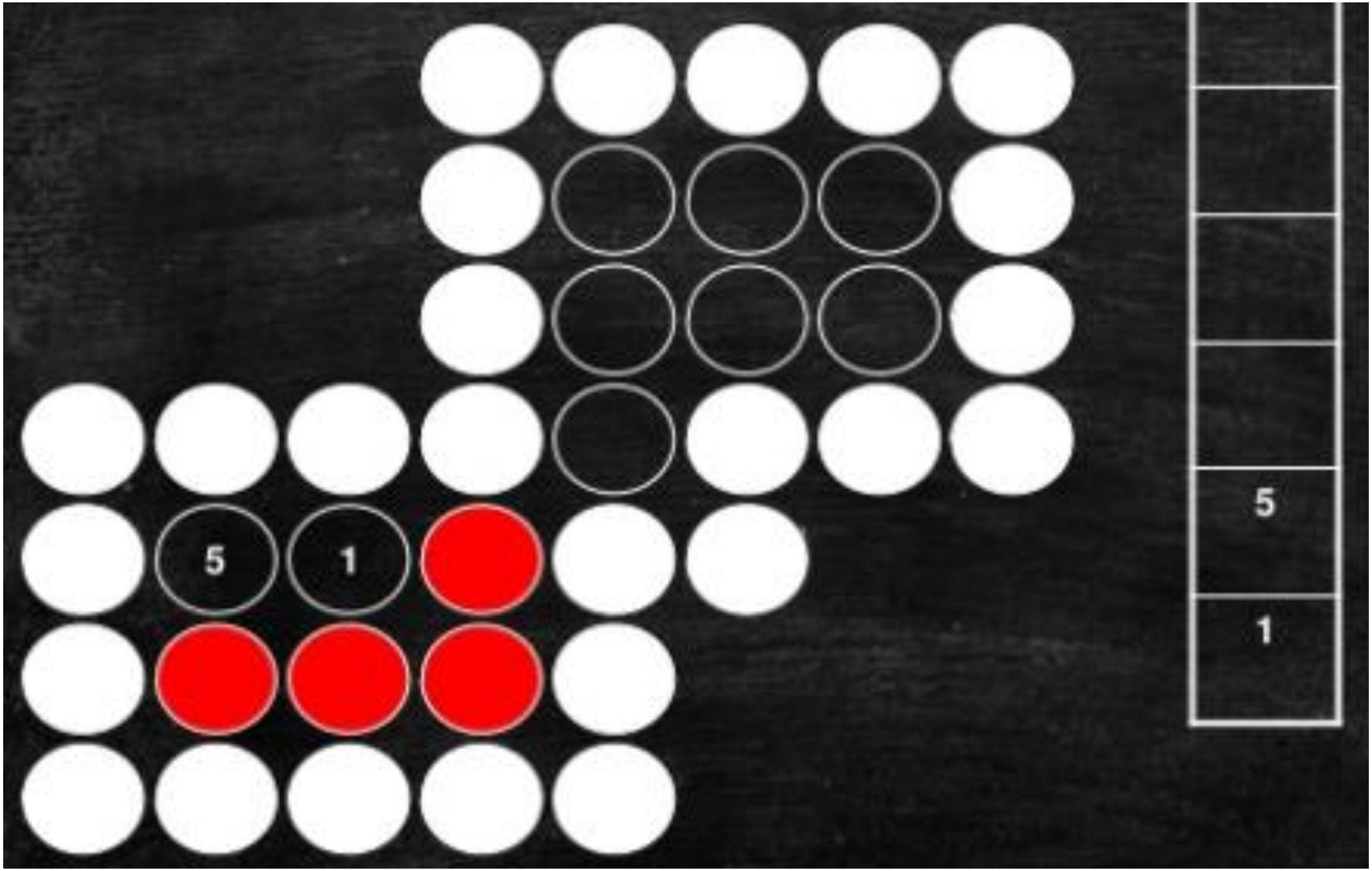
Boundary Fill Algorithms Example: 4-connected



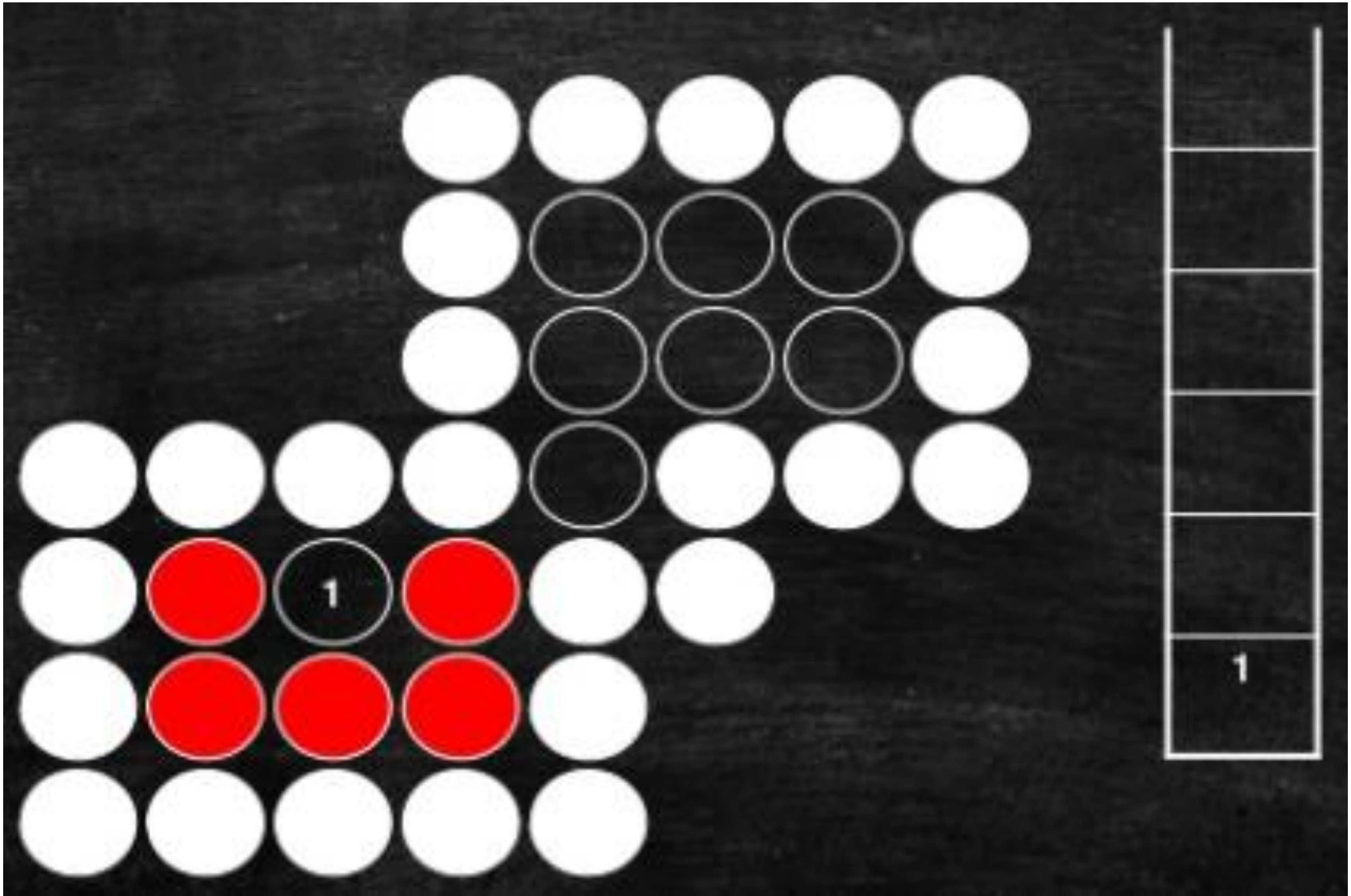
Boundary Fill Algorithms Example: 4-connected



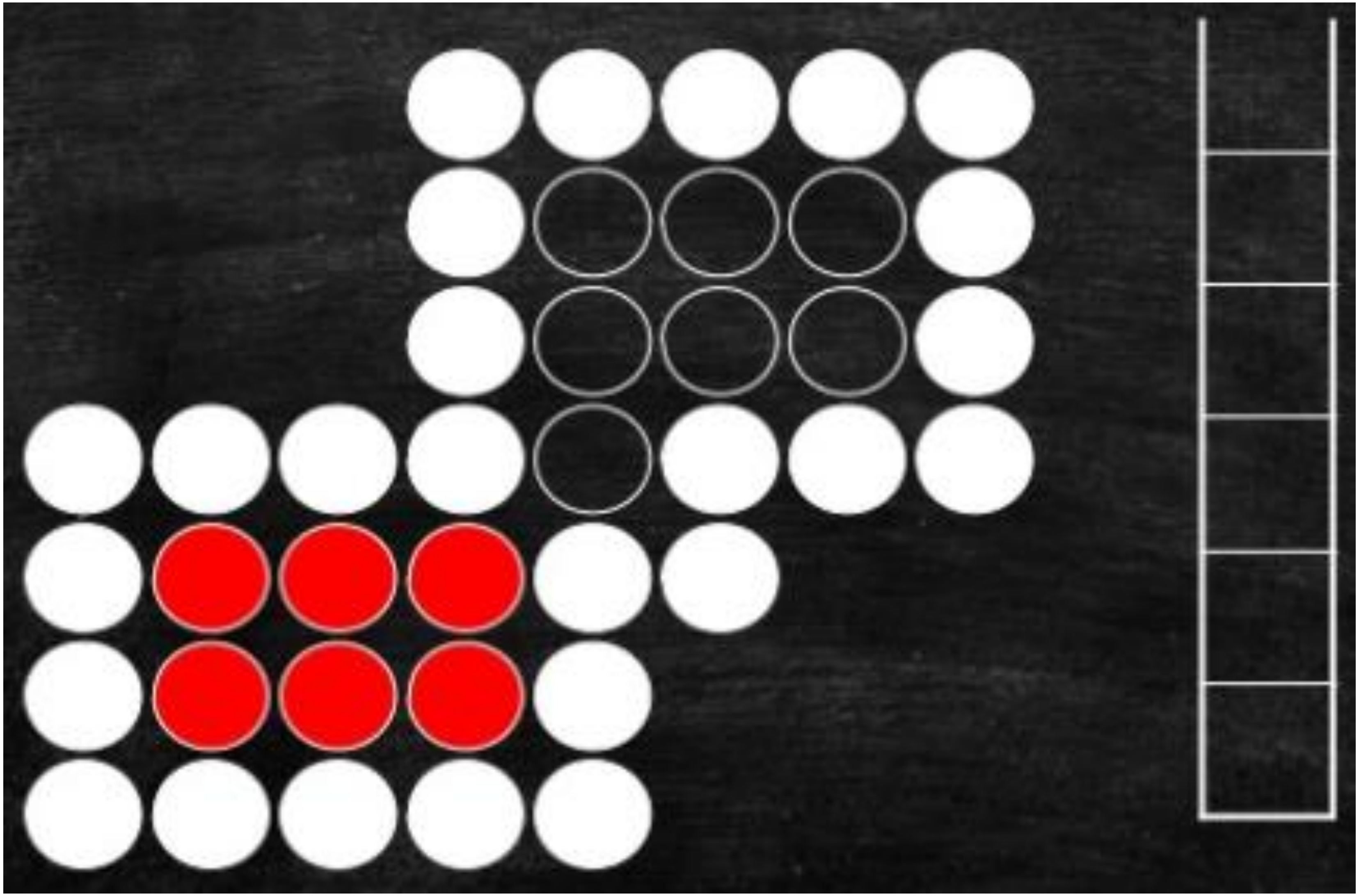
Boundary Fill Algorithms
Example: 4-connected



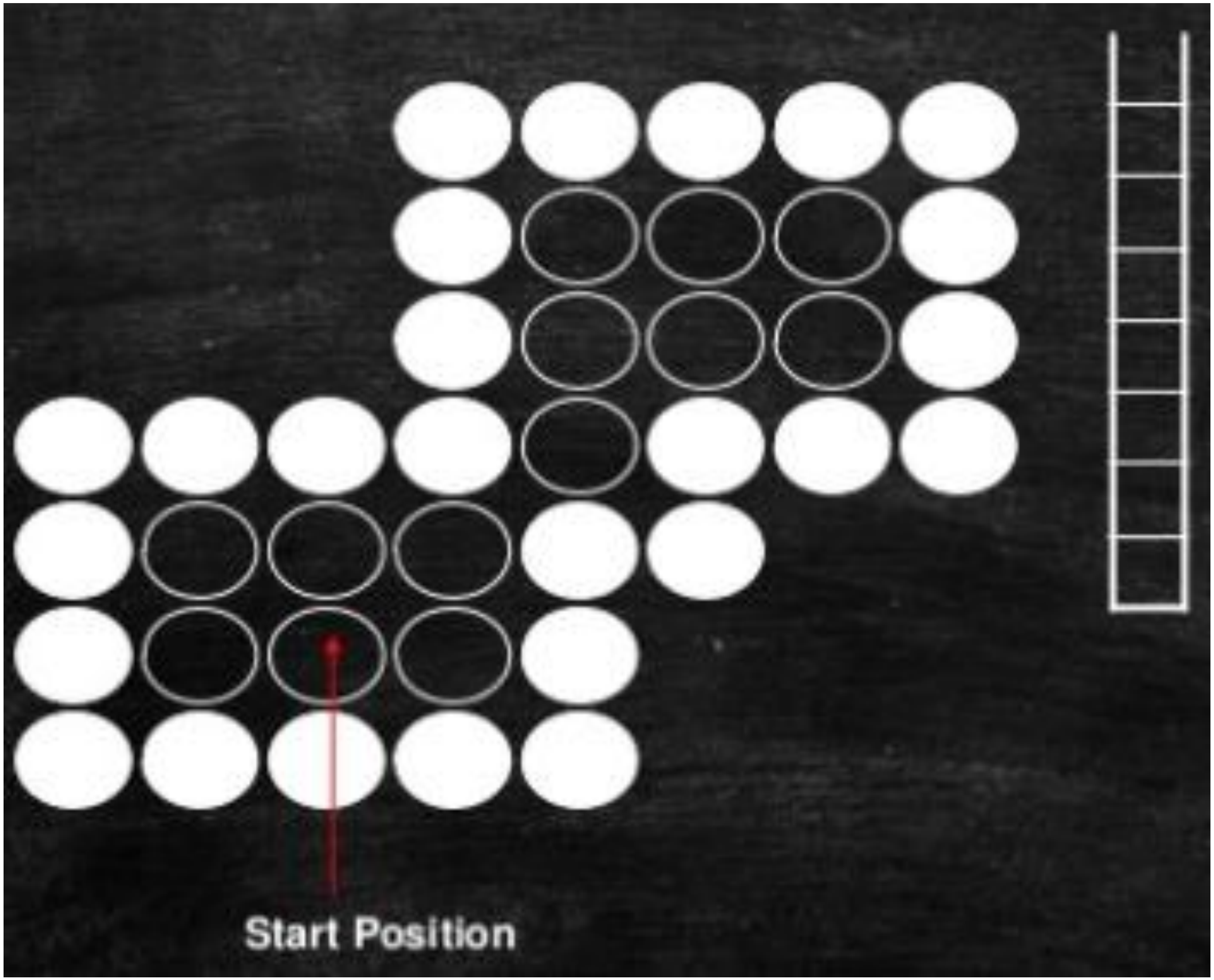
Boundary Fill Algorithms Example: 4-connected



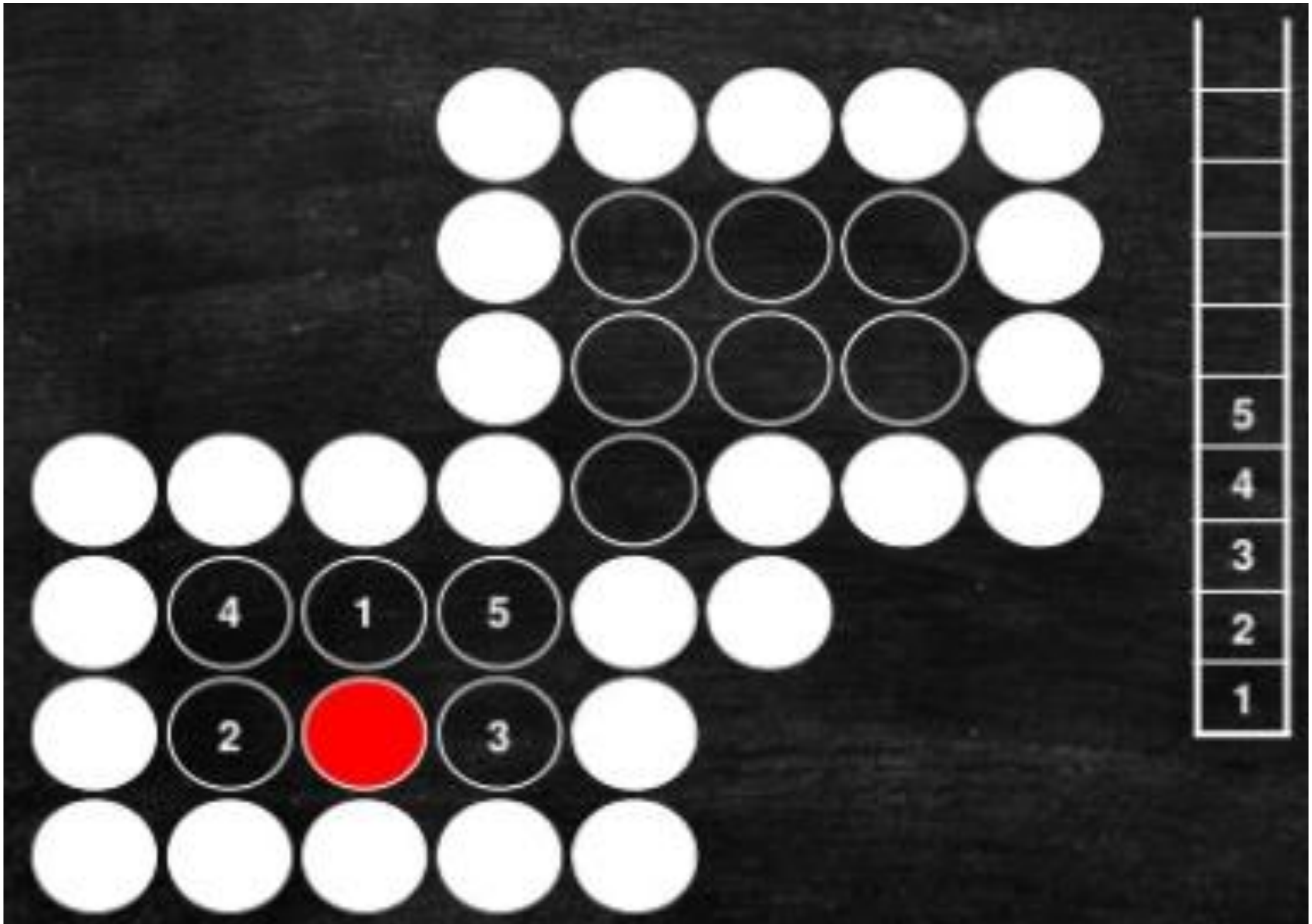
Boundary Fill Algorithms Example: 4-connected



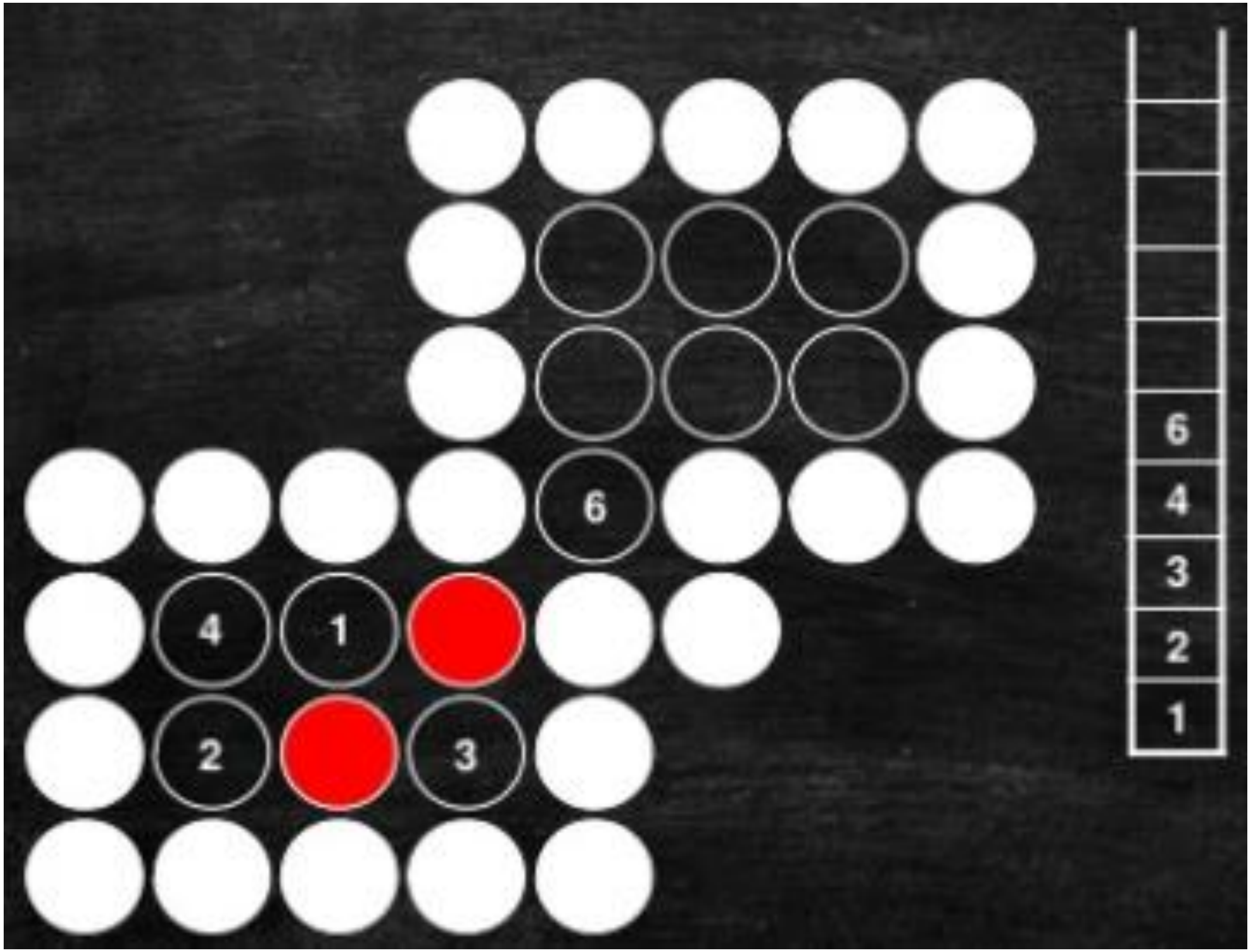
Boundary Fill Algorithms Example: 8-connected



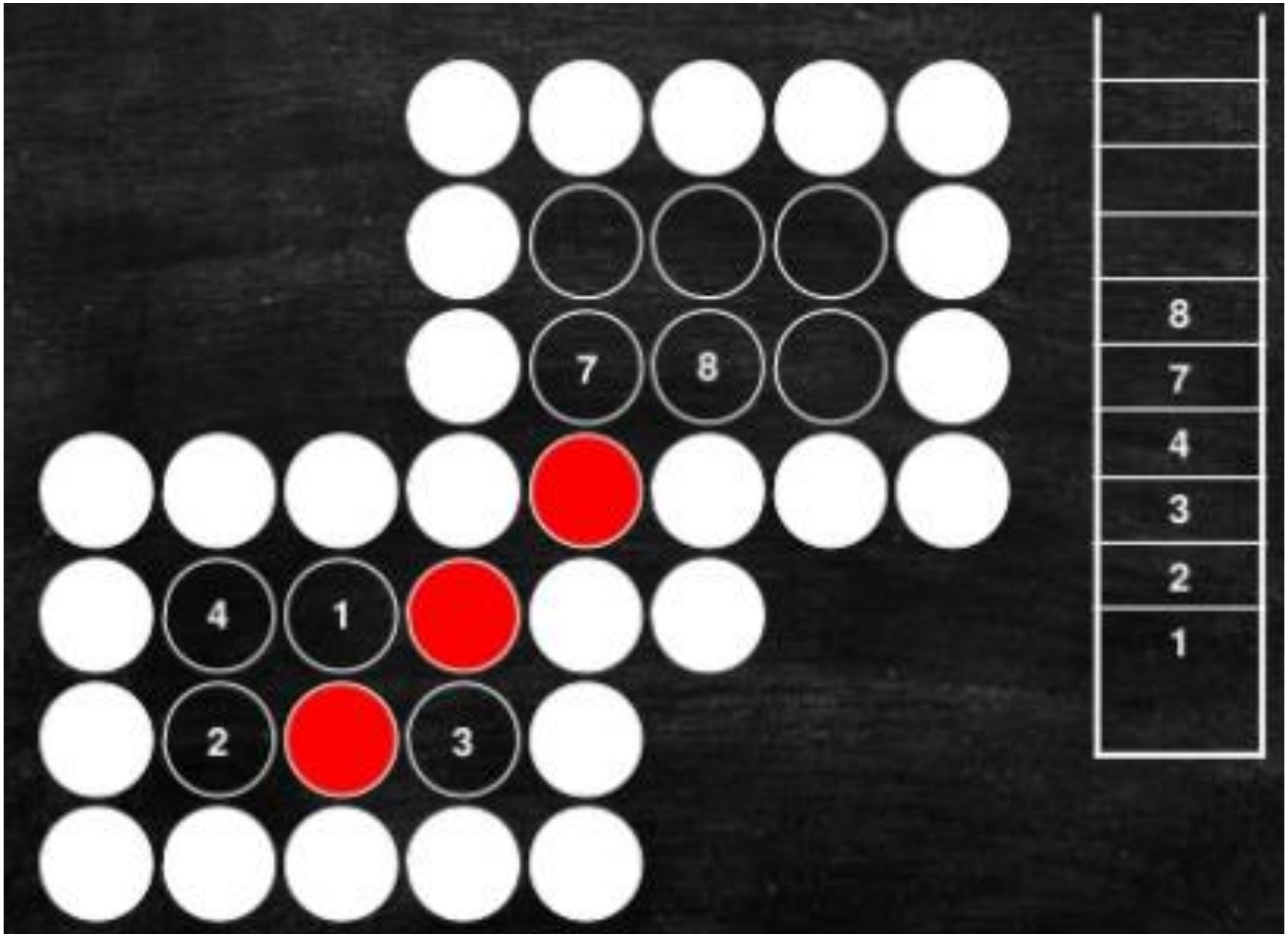
Boundary Fill Algorithms
Example: 8-connected



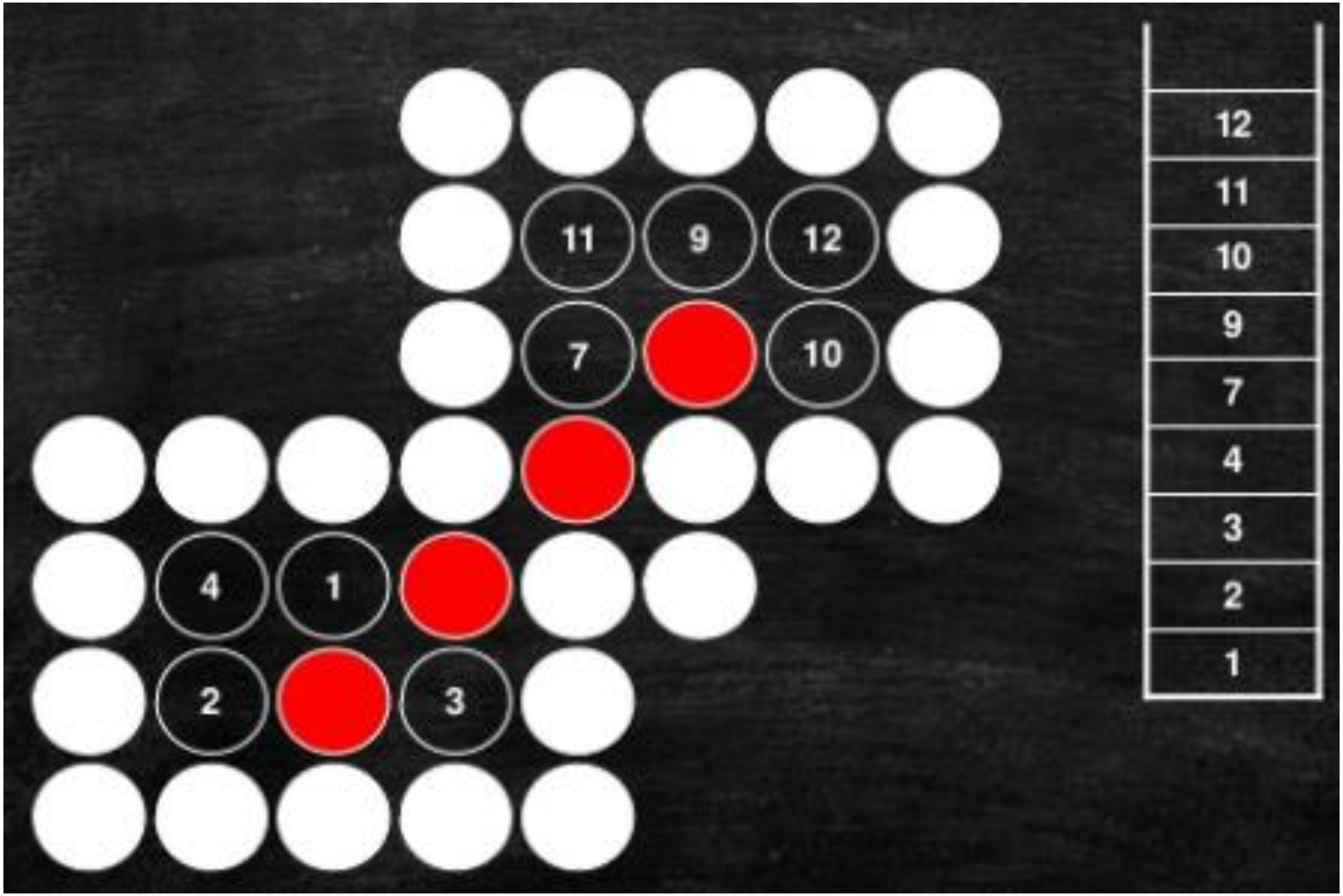
Boundary Fill Algorithms Example: 8-connected



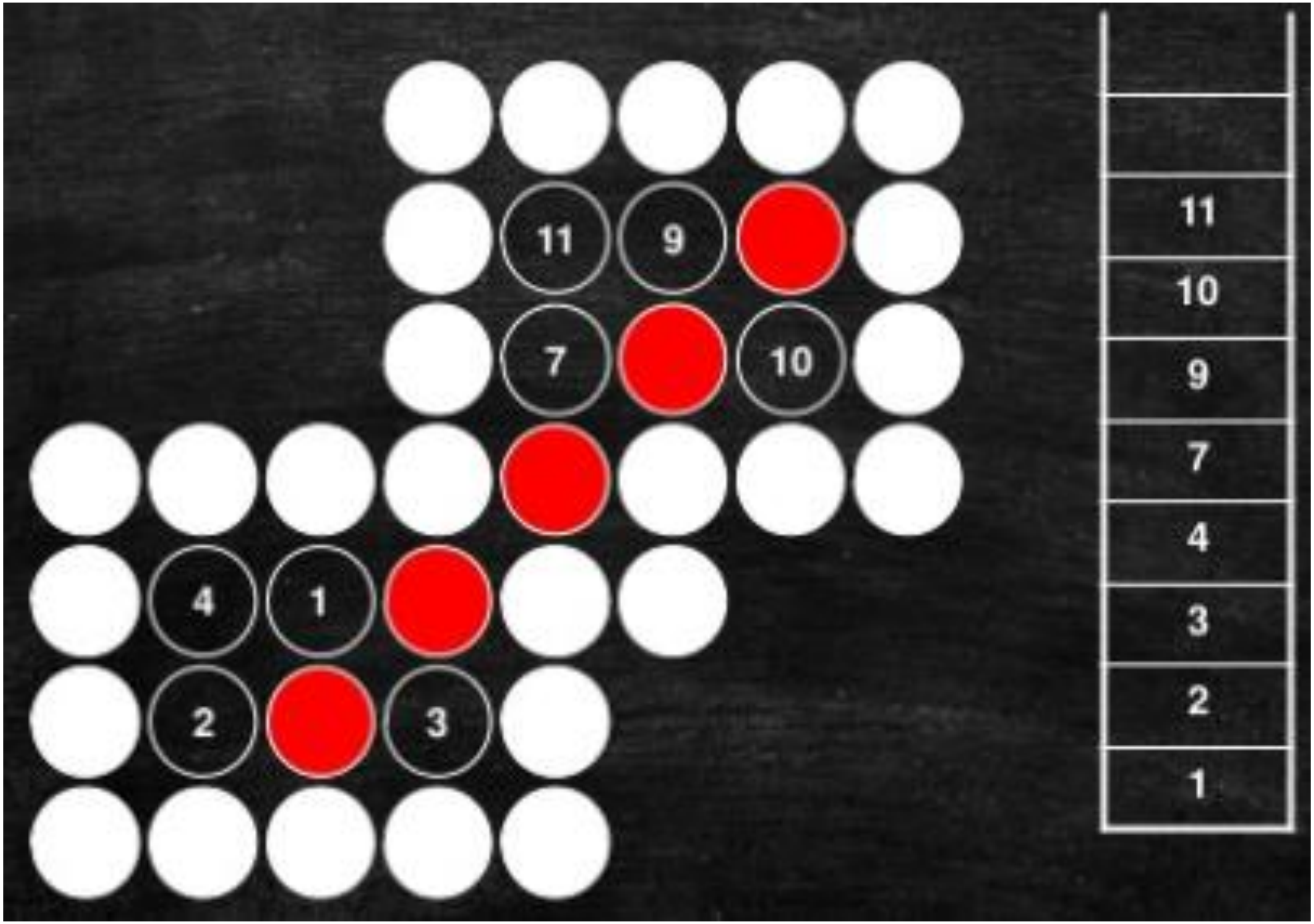
Boundary Fill Algorithms Example: 8-connected



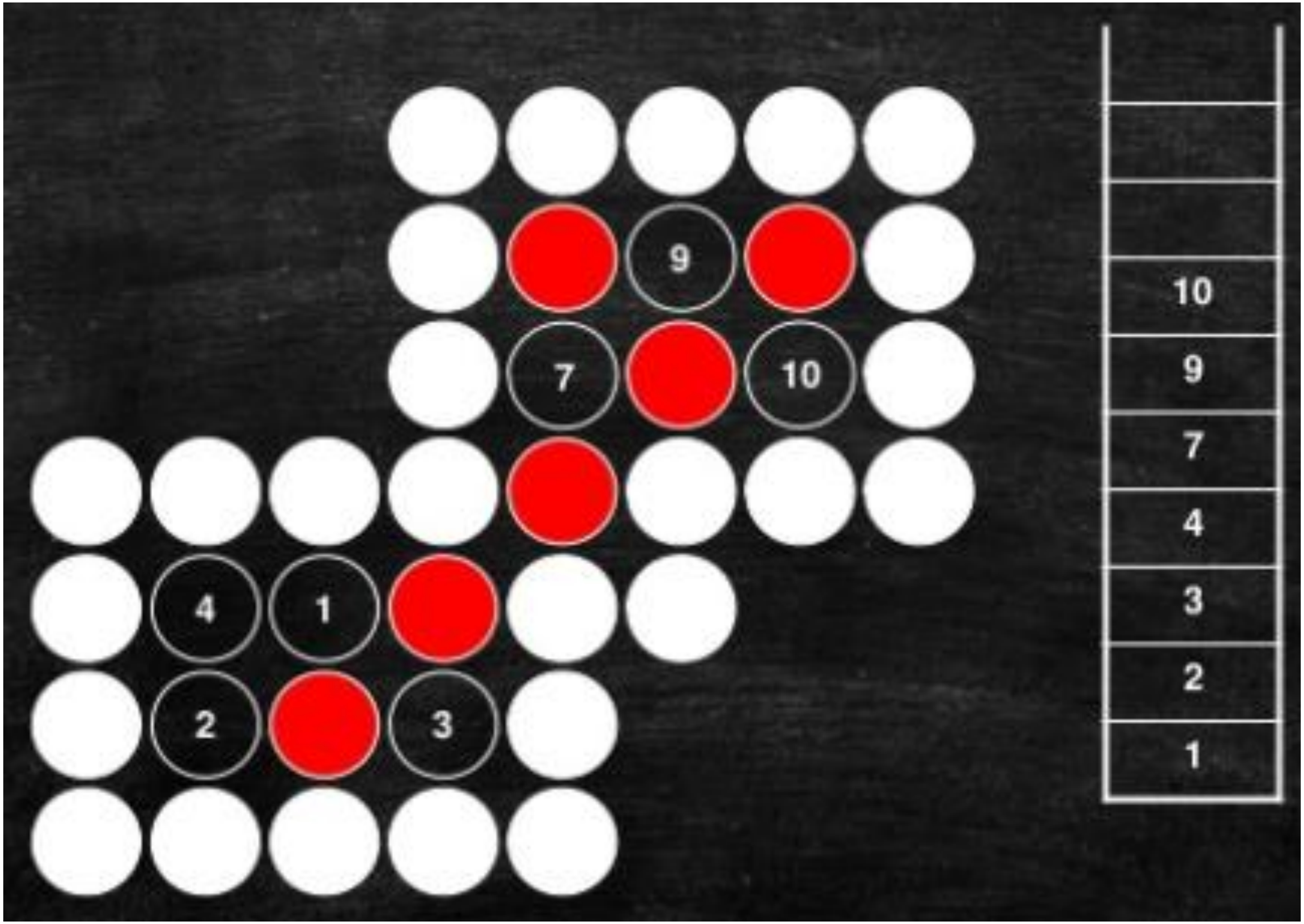
Boundary Fill Algorithms Example: 8-connected



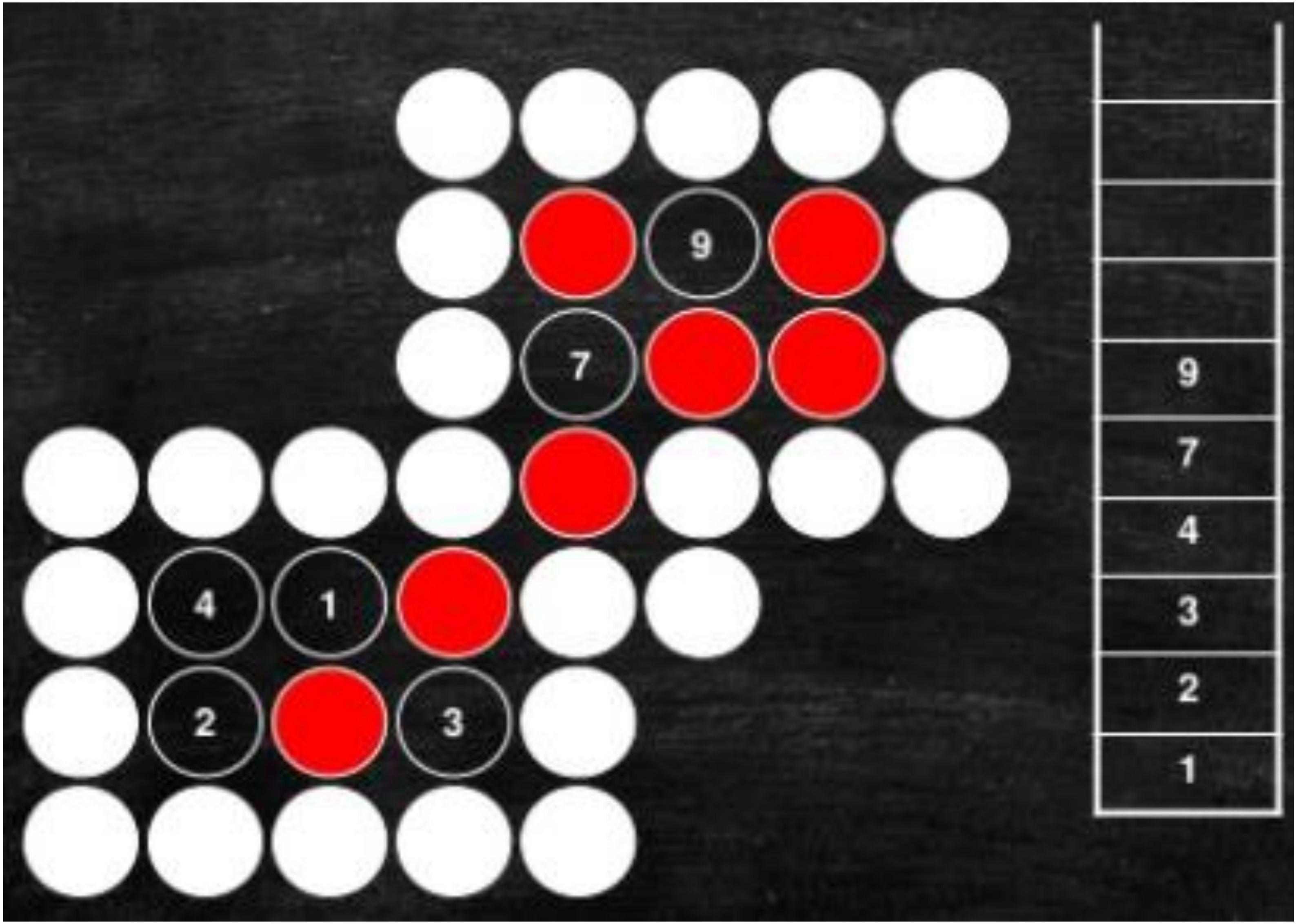
Boundary Fill Algorithms Example: 8-connected



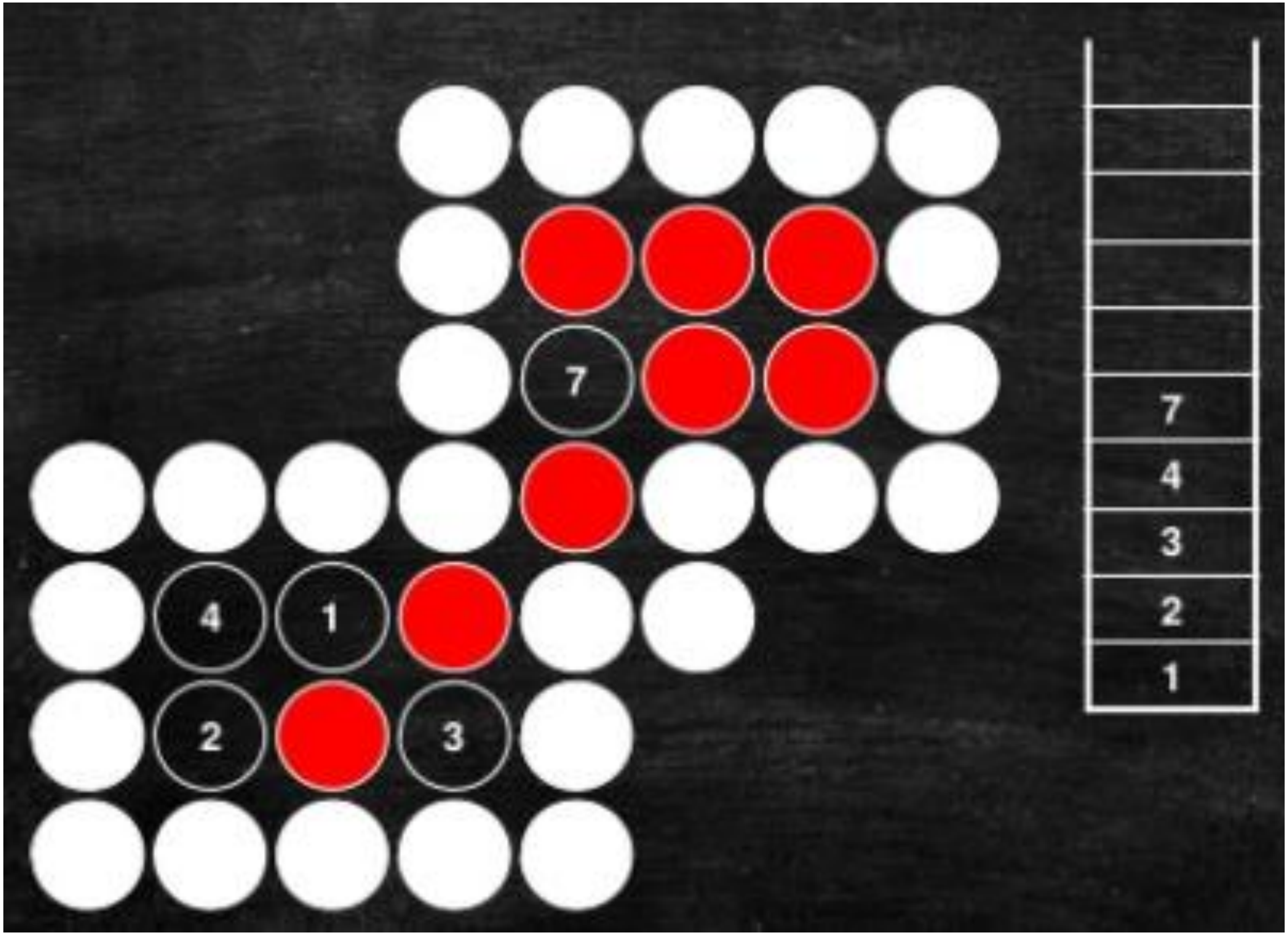
Boundary Fill Algorithms
Example: 8-connected



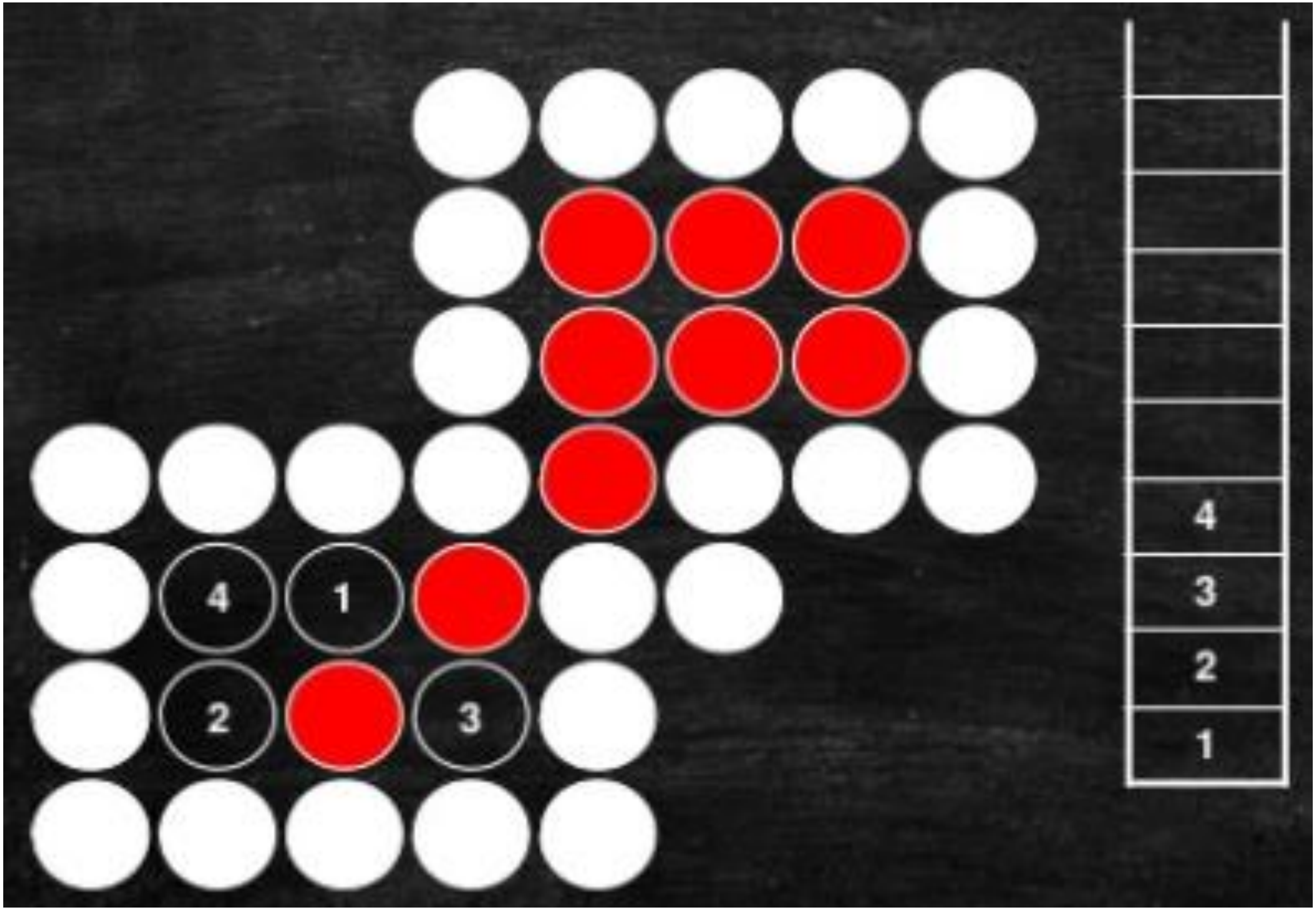
Boundary Fill Algorithms
Example: 8-connected



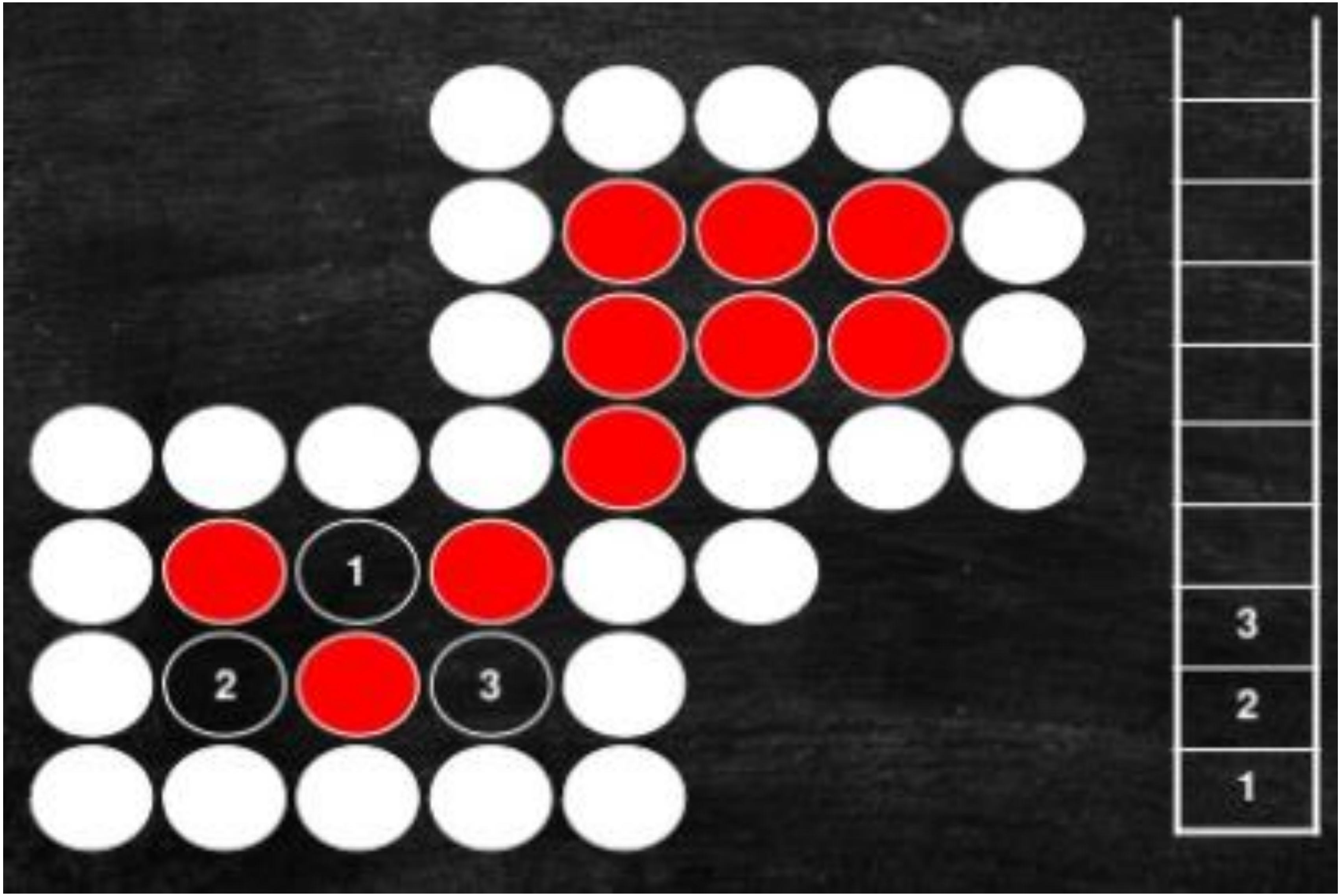
Boundary Fill Algorithms Example: 8-connected



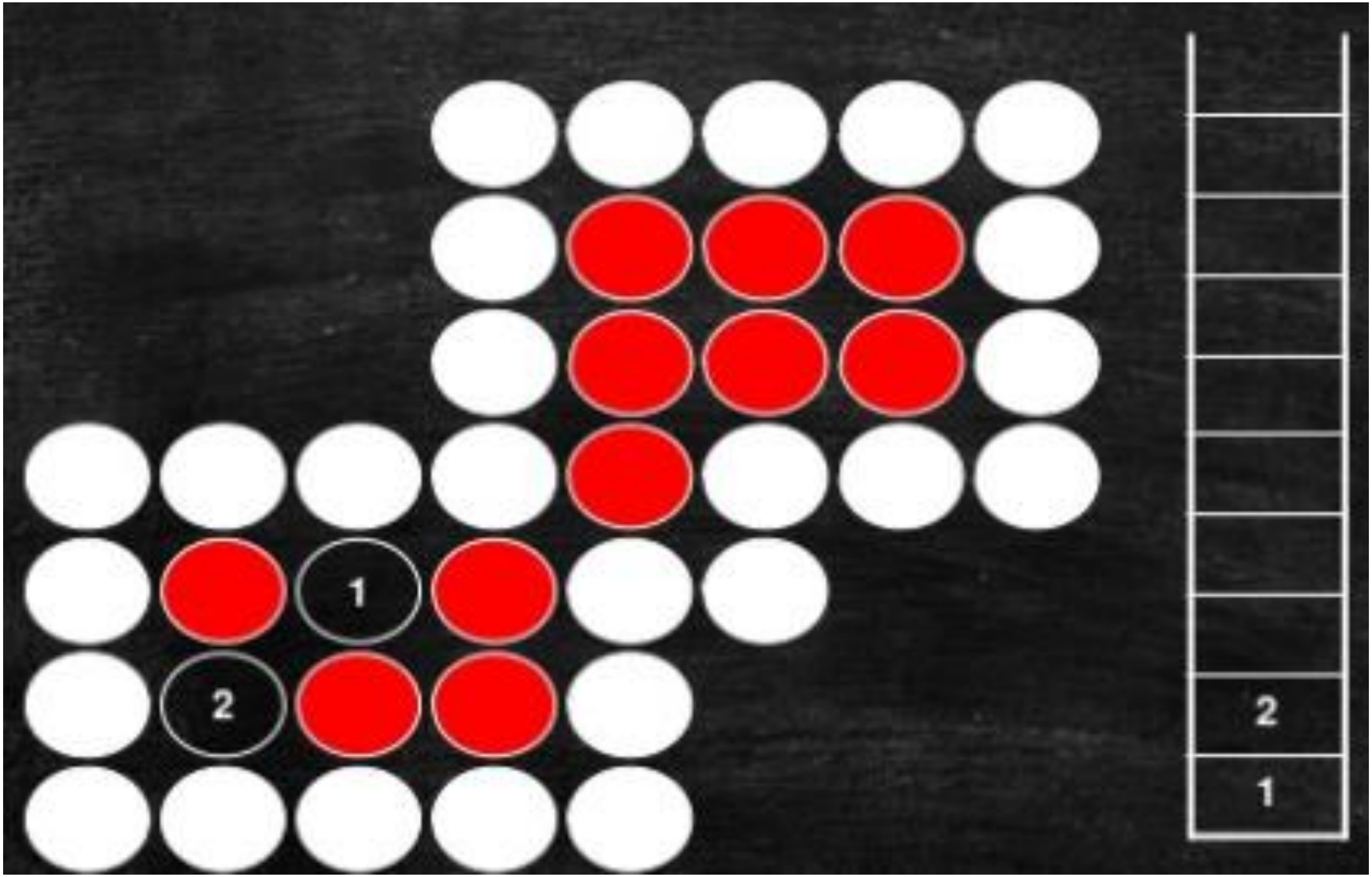
Boundary Fill Algorithms Example: 8-connected



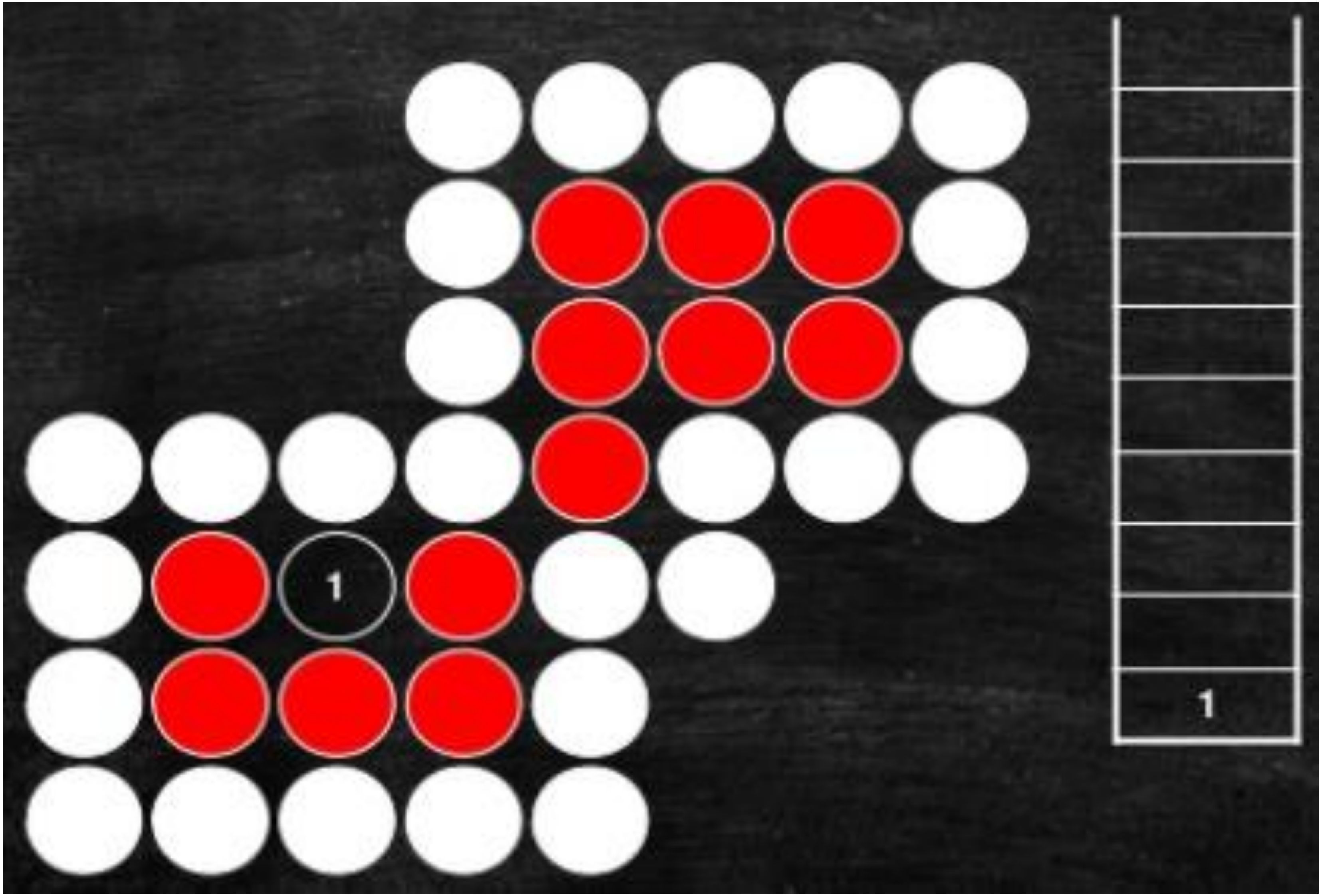
Boundary Fill Algorithms Example: 8-connected



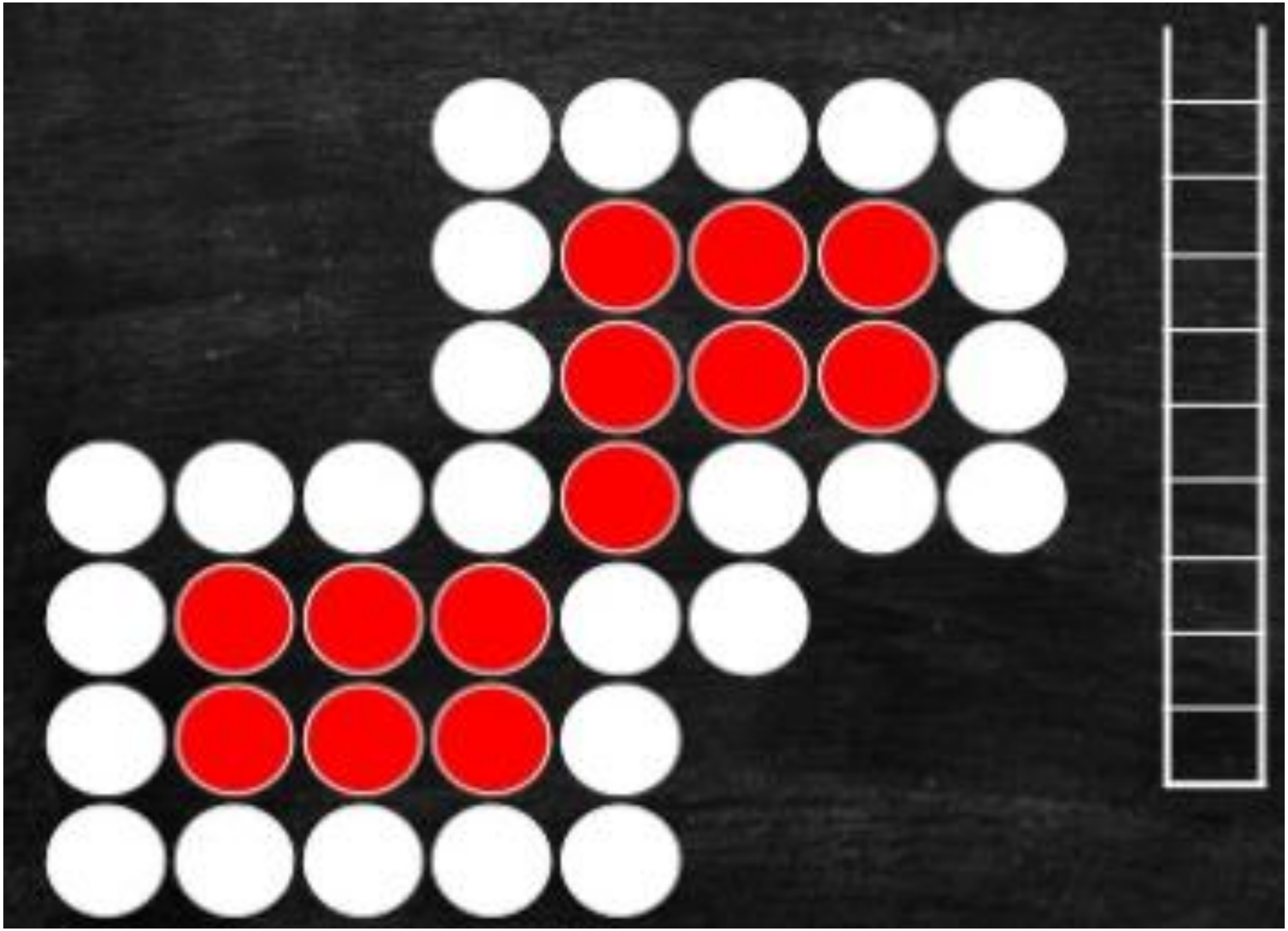
Boundary Fill Algorithms
Example: 8-connected



Boundary Fill Algorithms
Example: 8-connected



Boundary Fill Algorithms Example: 8-connected



Flood Fill Algorithms

- To fill an area or recolor it whose boundary is not defined by a single color
- Paint by replacing color instead of checking for boundary color
- If more than one interior color, first reassign to a single color
- 4-connected or 8-connected approach

```
void floodFill4 (int x, int y, int fillColor, int oldColor)
{
  if (getPixel (x, y) == oldColor) {
    setColor (fillColor);
    setPixel (x, y);
    floodFill4 (x+1, y, fillColor, oldColor);
    floodFill4 (x-1, y, fillColor, oldColor);
    floodFill4 (x, y+1, fillColor, oldColor);
    floodFill4 (x, y-1, fillColor, oldColor);
  }
}
```

Flood Fill Algorithms