Segmentation

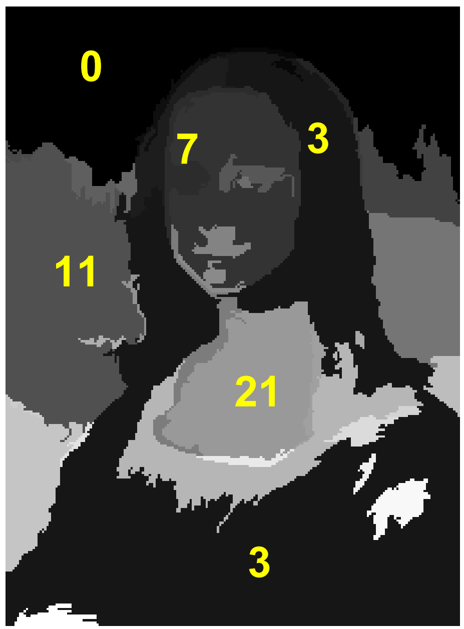
ECE 847:
Digital Image Processing

Stan Birchfield
Clemson University

What is segmentation?

- Segmentation divides an image into groups of pixels
- Pixels are grouped because they share some local property (gray level, color, texture, motion, etc.)


boundaries



labels


pseudocolors


mean colors

(different ways of displaying the output)

algorithm used: Pedro F. Felzenszwalb and Daniel P. Huttenlocher, Efficient Graph-Based Image Segmentation, IJCV, 59(2), 2004

Other variants

- *Segmentation = partitioning*
  Carve dense data set into (disjoint) regions
  - Divide image based on pixel similarity
  - Divide spatiotemporal volume based on image similarity (shot detection)
  - Figure / ground separation (background subtraction)
  - Regions can be overlapping (layers)

- *Grouping = clustering*
  Gather sets of items according to some model
  - If items are dense, then essentially the same problem as above (e.g., clustering pixels)
  - If items are sparse, then problem has a slightly different flavor:
    - Collect tokens that lie on a line (robust line fitting)
    - Collect pixels that share the same fundamental matrix (independent 3D rigid motion)
    - Group 3D surface elements that belong to the same surface

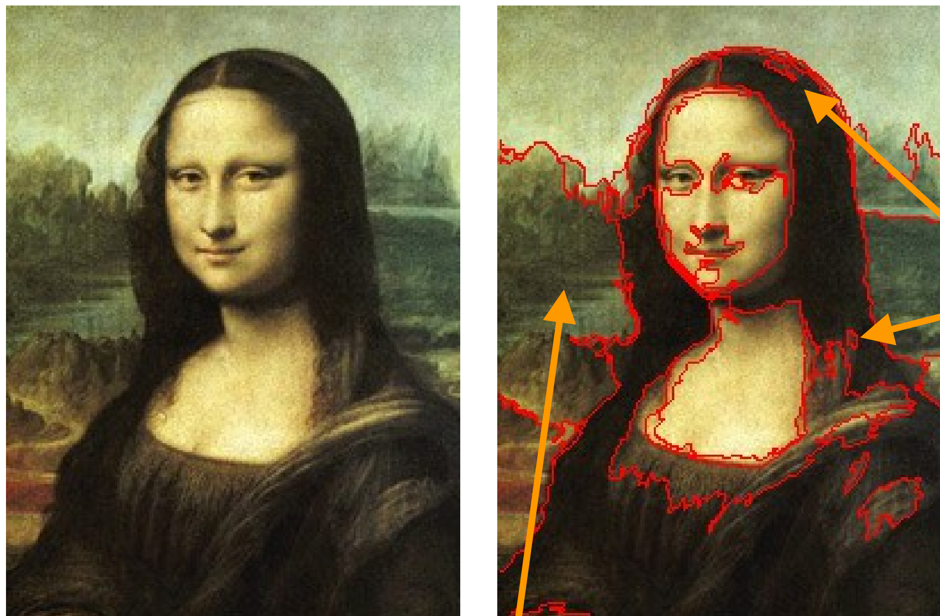
The problems are closely related, but we will treat sparse clustering in a separate lecture (model fitting)

Foreground / background separation

Background subtraction provides figure-ground separation, which is a type of segmentation

C. Wren et al., Pfinder: Real-Time Tracking of the Human Body
http://vismod.media.mit.edu/vismod/demos/pfinder/

Two errors



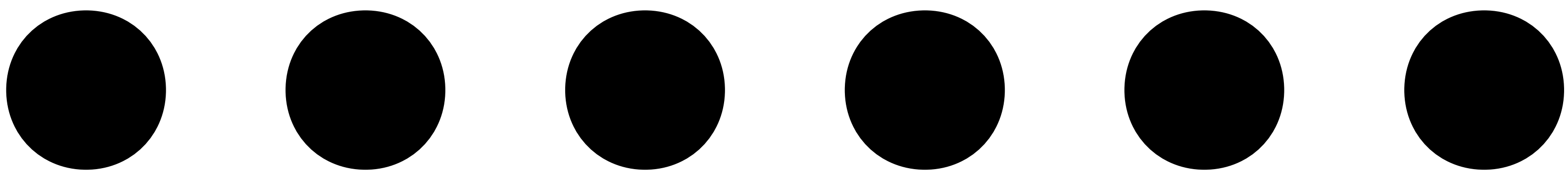
oversegmentation
(hair should
be one group)

undersegmentation
(water should be
separated from trees)

Outline

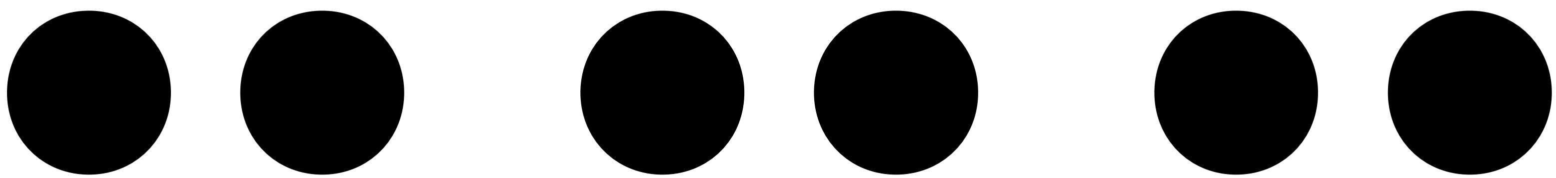
- Human segmentation
- Standard algorithms:
  - Split-and-merge
  - Region growing
  - Minimum spanning tree
- Watershed algorithm
- Normalized cuts

An experiment:
What do you see?



Just six dots

Now what do you see?

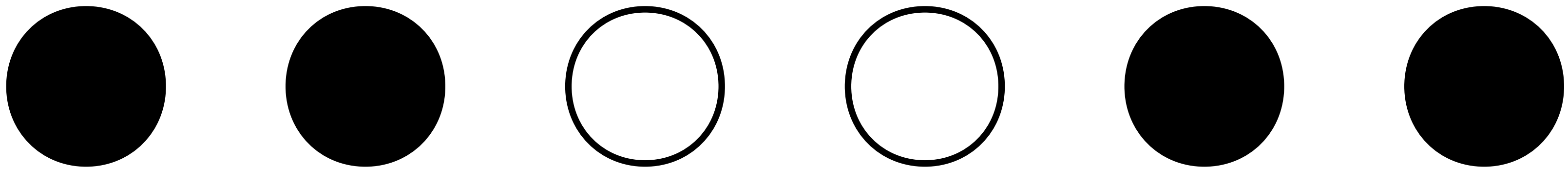


Three groups of dot pairs

Why?

Dots that are close together ("proximity")
are grouped together by the human visual system

And now?

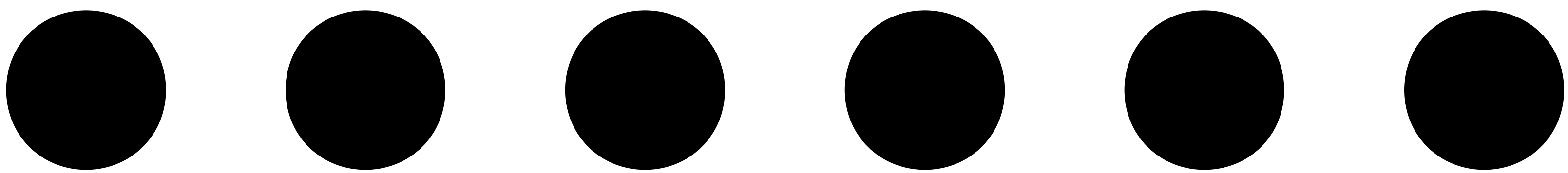


Again, three groups of dot pairs

Why?

Dots are similar in appearance (“similarity”)

How about now?

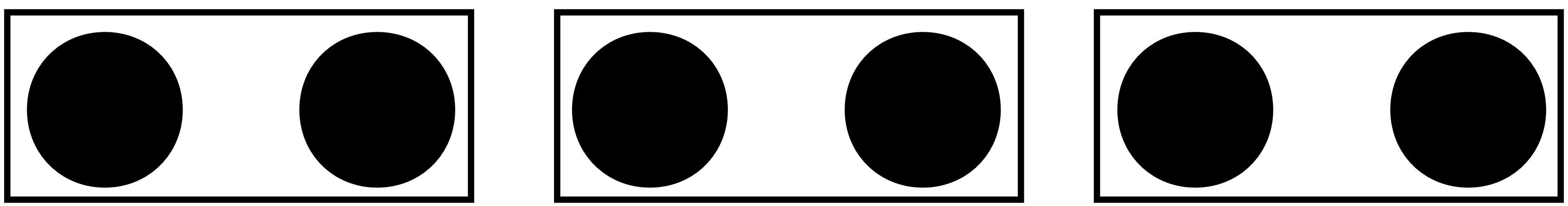


Again, three groups of dot pairs

Why?

Dots move similarly (“common fate”)

Last one

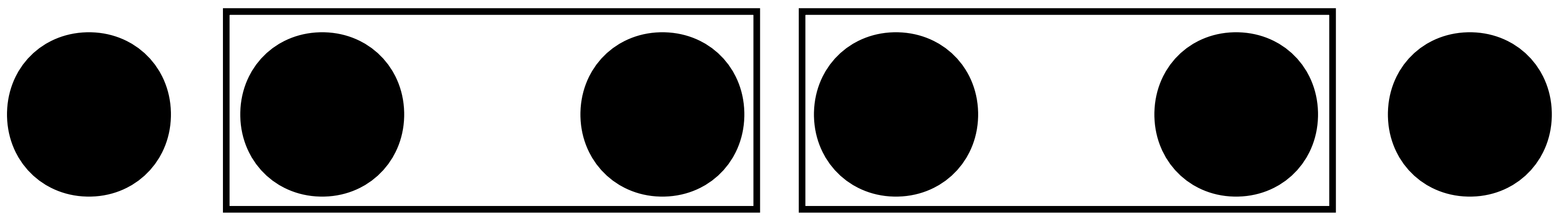


Again, three groups of dots

Why?

Dots are enclosed together ("common region")

But wait!



Note that the "common region" can overwhelm the "proximity" tendency

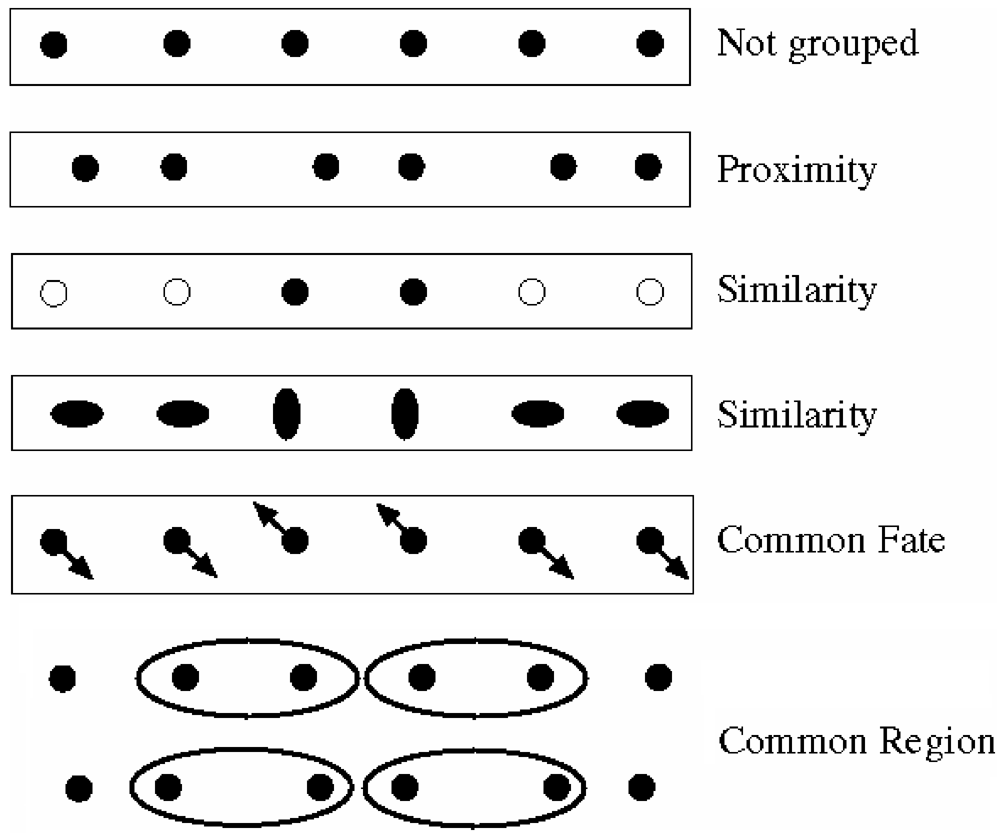
Gestalt psychology

Gestalt school of psychologists emphasized grouping as the key to understanding visual perception.

Recall: Context affects how things are perceived

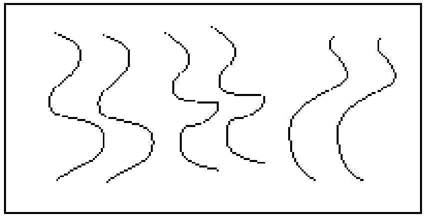
gestalt **– whole or group**

gestalt qualitat **– set of internal relationships that makes it a whole**

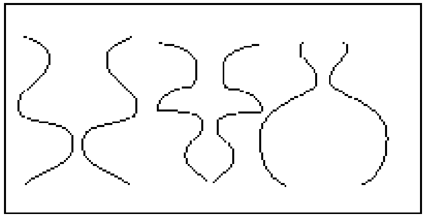


Max Wertheimer, Laws of Organization in Perceptual Forms, 1923
http://psy.ed.asu.edu/~classics/Wertheimer/Forms/forms.htm

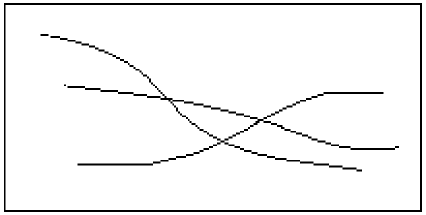
Gestalt psychology (cont.)



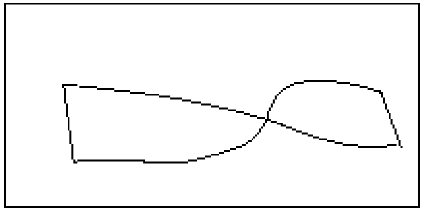
Parallelism



Symmetry

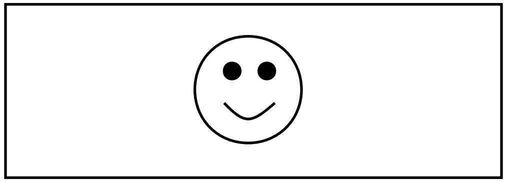


Continuity

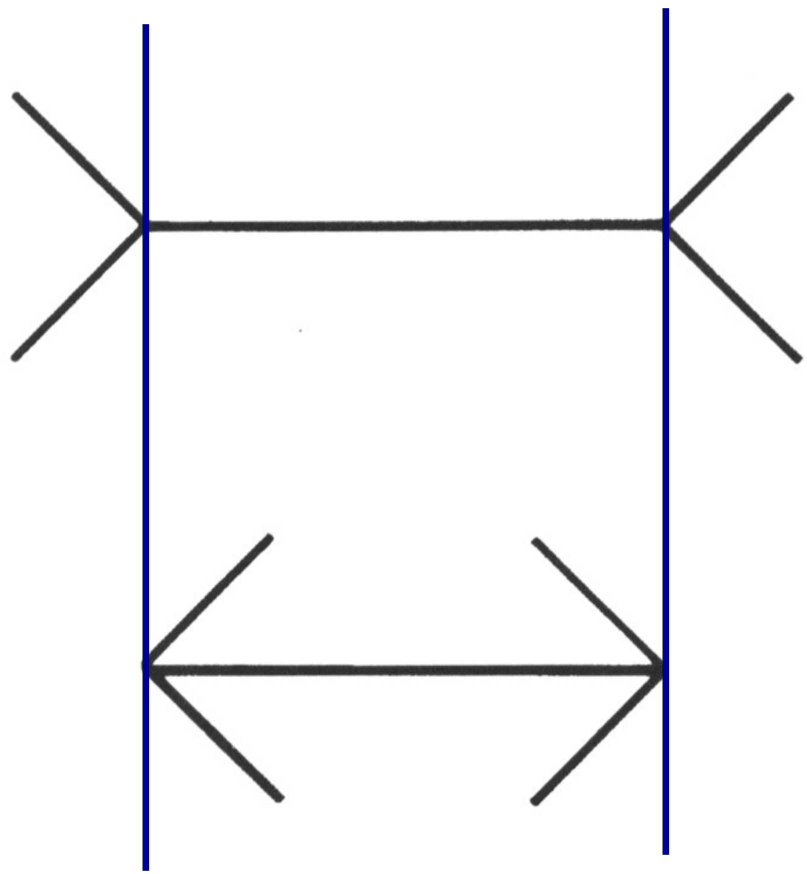


Closure

Familiar configuration

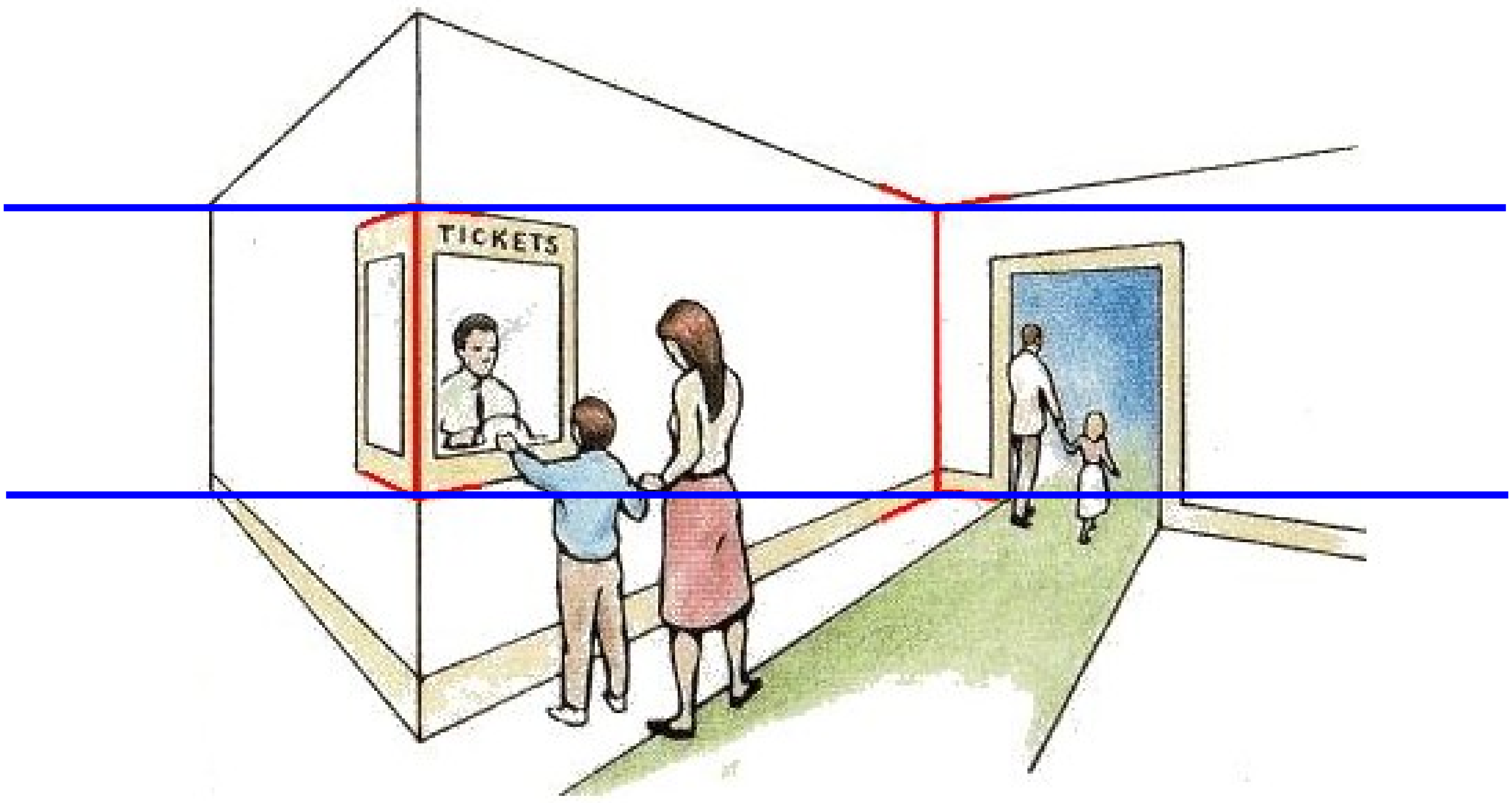


Muller-Lyer illusion



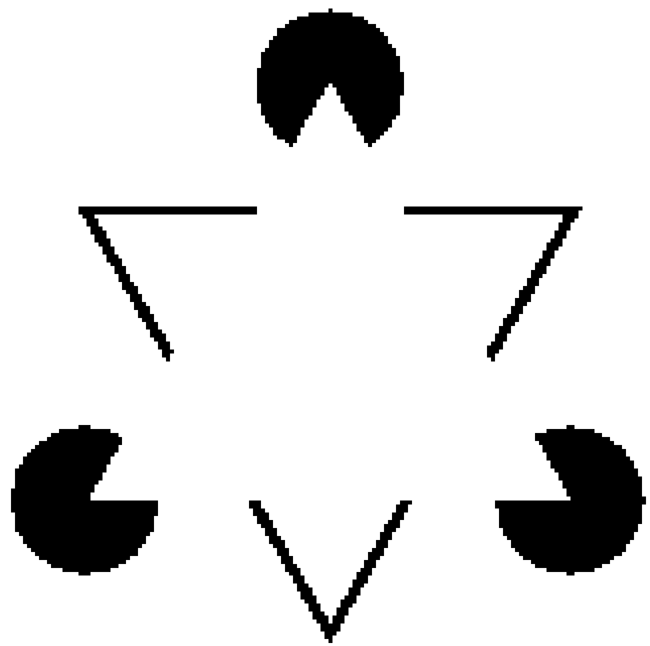
Lines are perceived as components of a whole rather than as individual lines.

3D interpretation of Muller-Lyer



from http://www.michaelbach.de/ot/sze_muelue/

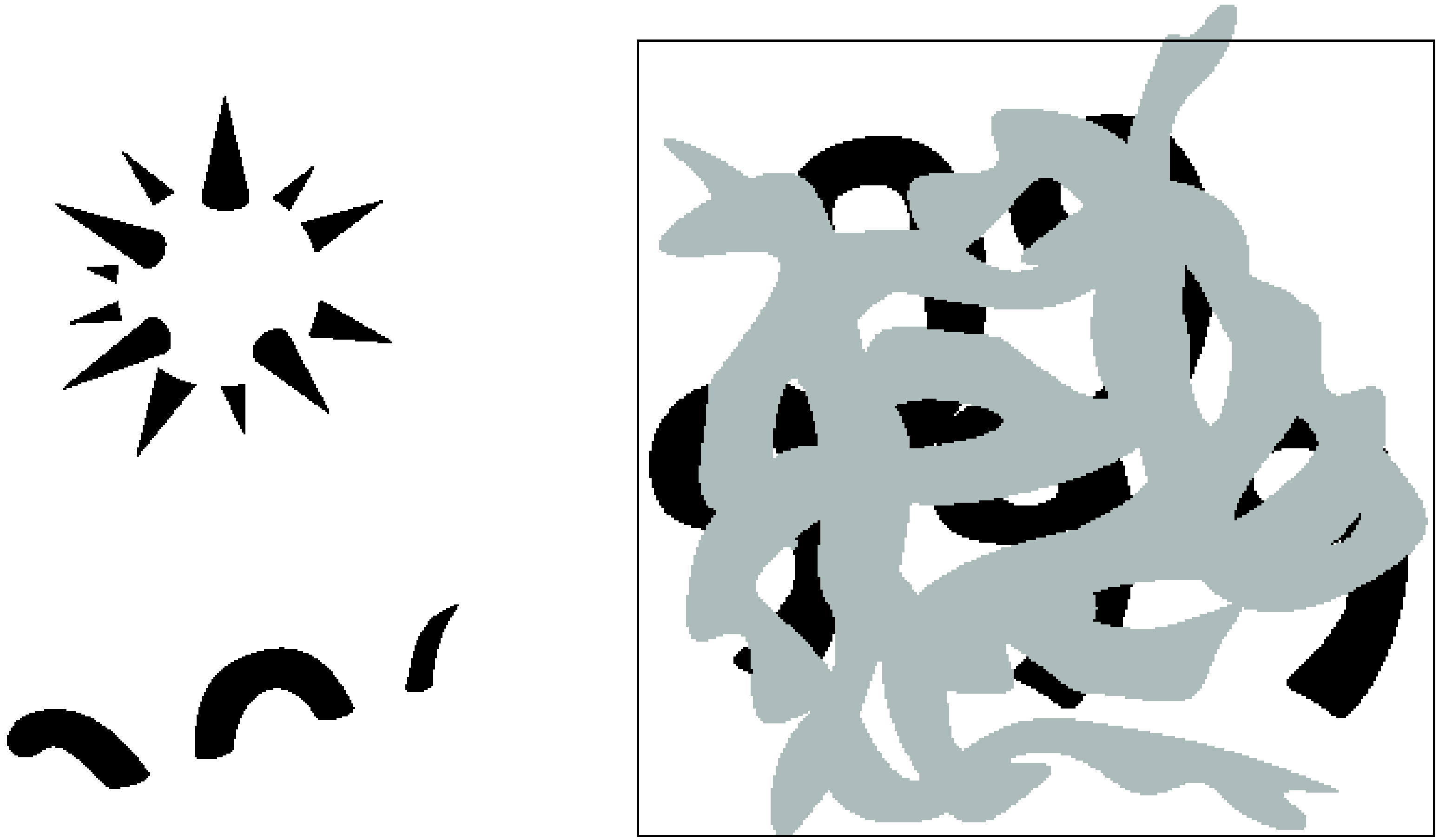
Can you see anything invisible?



These are **illusory contours**, formed by grouping the circles

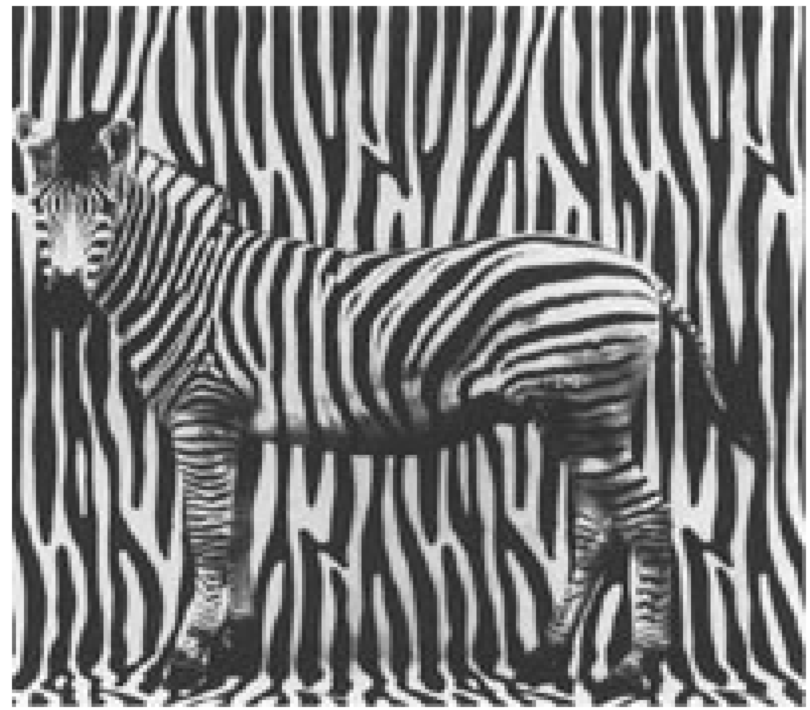
This is the well-known Kanizsa triangle

More illusory contours



Grouping by invisible completions

Two final examples



What role is top-down playing?

from http://www-static.cc.gatech.edu/classes/AY2007/cs4495_fall/html/materials.html

Back to computer vision…

Outline

- Human segmentation
- Standard algorithms:
  - Split-and-merge
  - Region growing
  - Minimum spanning tree
- Watershed algorithm
- Normalized cuts

Segmentation as partitioning

- A *partition* of image is collection of sets $S_1, .., S_N$ such that

  $I = S_1 \cup S_2 \ldots \cup S_N$ (sets cover entire image)

  $S_i \cap S_j = 0$ for all $i \neq j$ (sets do not overlap)

- A *predicate* $H(S_i)$ measures region *homogeneity*

$$H(R) = \begin{cases} \textit{true} & \text{if pixels in region } R \text{ are similar} \\ \textit{false} & \text{otherwise} \end{cases}$$

- We want
  1. Regions to be homogeneous

     $H(S_i) = \textit{true}$ for all i

  2. Adjacent regions to be different from each other

     $H(S_i \cup S_j) = \textit{false}$ for all adjacent S_i, S_j

Two approaches

- **Splitting**
  (Divisive clustering)
  - start with single region covering entire image
  - repeat: split inhomogeneous regions
  - even better:
    repeat: split cluster to yield two distant components (difficult)

Property 2 is always true:
$H(S_i \cup S_j)$ = *false* for adjacent regions

Goal is to satisfy Property 1:
$H(S_i)$ = *true* for every region

- **Merging**
  (Agglomerative clustering)
  - start with each pixel as a separate region
  - repeat: merge adjacent regions if union is homogeneous
  - even better:
    repeat: merge two closest clusters

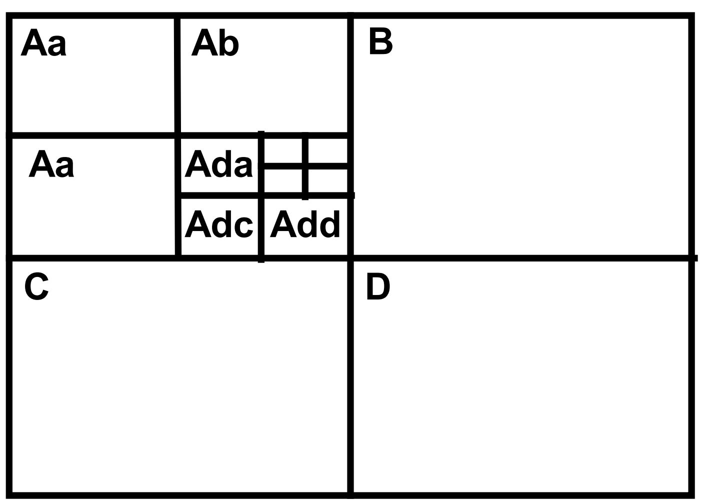
Property 1 is always true:
$H(S_i)$ = *true* for every region

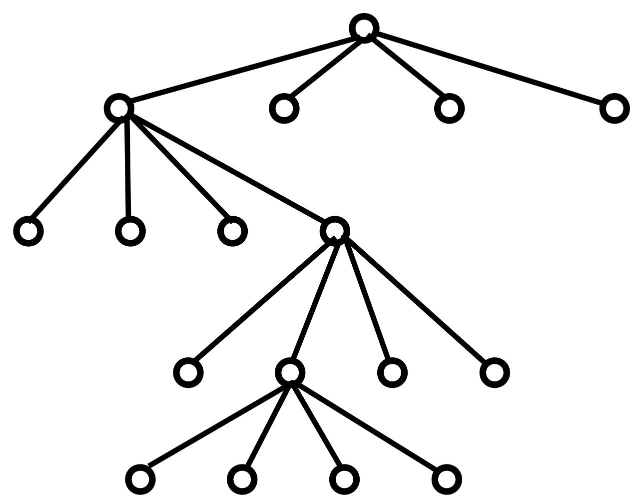
Goal is to satisfy Property 2:
$H(S_i \cup S_j)$ = *false* for adjacent regions

In practice, merging works much better than splitting

Region splitting

- Start with entire image as a single region
- Repeat:
  - Split any region that does not satisfy homogeneity criterion into subregions
- Quad-tree representation is convenient
- Then need to merge regions that have been split

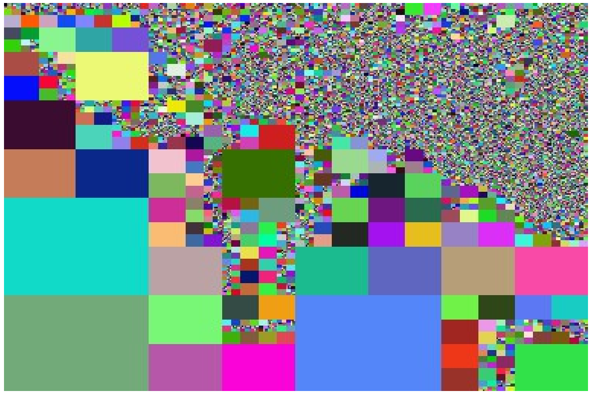




Split-and-Merge

- **Split-and-merge algorithm combines these two ideas**
  - **Split image into quadtree, where each region satisfies homogeneity criterion**
  - **Merge neighboring regions if their union satisfies criterion (like connected components)**

image



after split

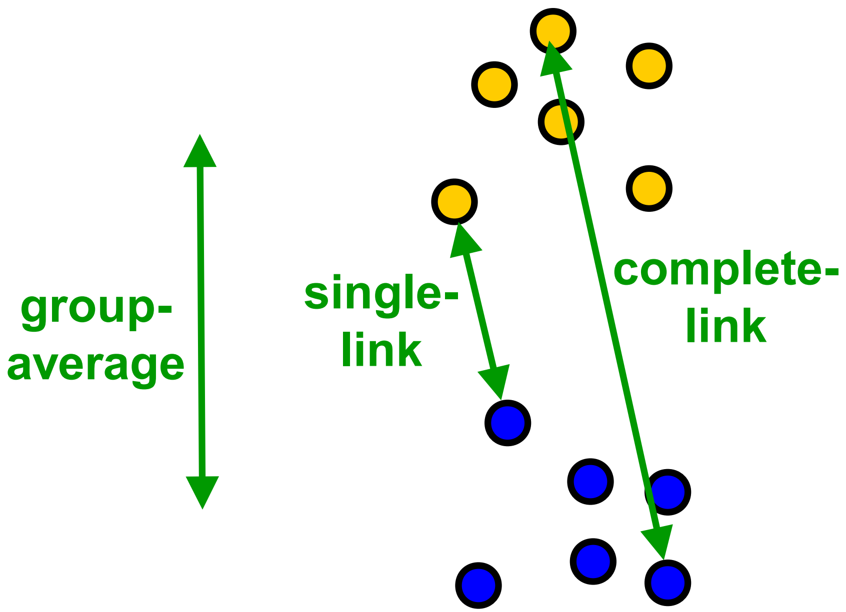
after merge

S. L. Horowitz and T. Pavlidis, Picture Segmentation by a Tree Traversal Algorithm, 1976

When to merge two clusters

Inter-cluster distance can be computed by

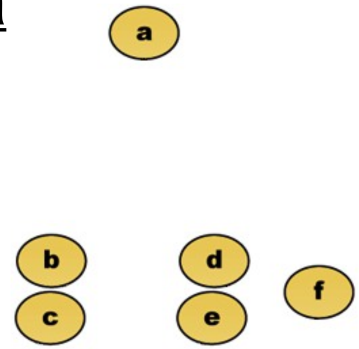
- single-link clustering
  (dist. b/w closest elements)
  allows adaptation
- complete-link clustering
  (dist. b/w farthest elements)
  avoids drift
- group-average clustering
  (use average distance)
  good compromise
- root clustering
  (dist. b/w initial points of clusters)
  variation on complete-link

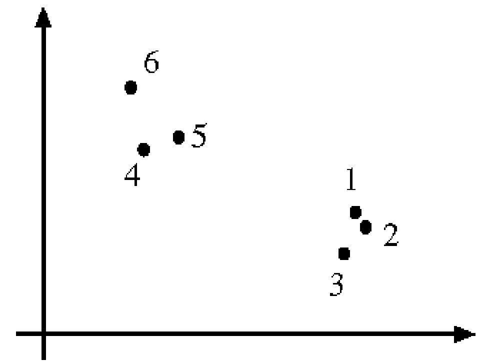


Dendrograms

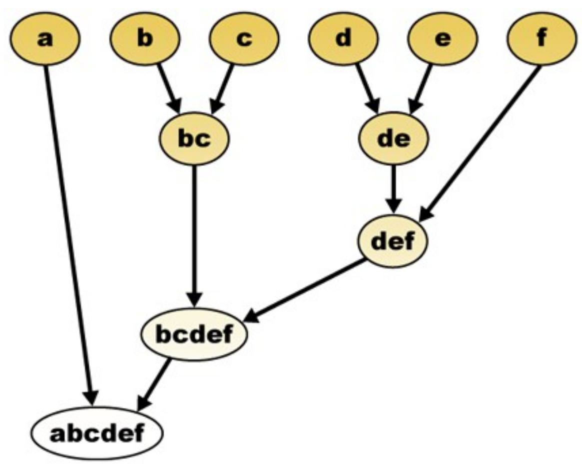
Dendrogram yields a picture of output as clustering process continues

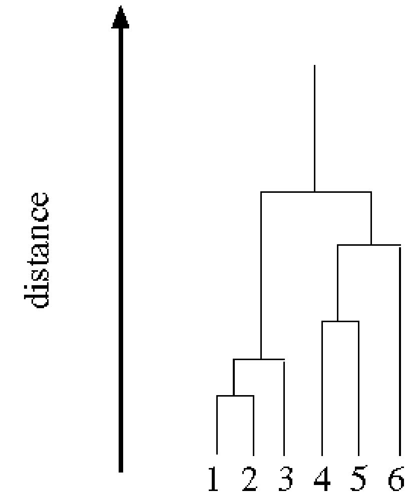
raw data





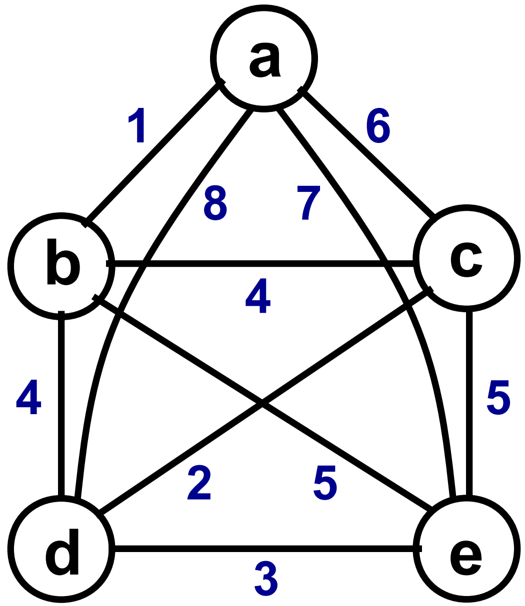
clusters represented as tree





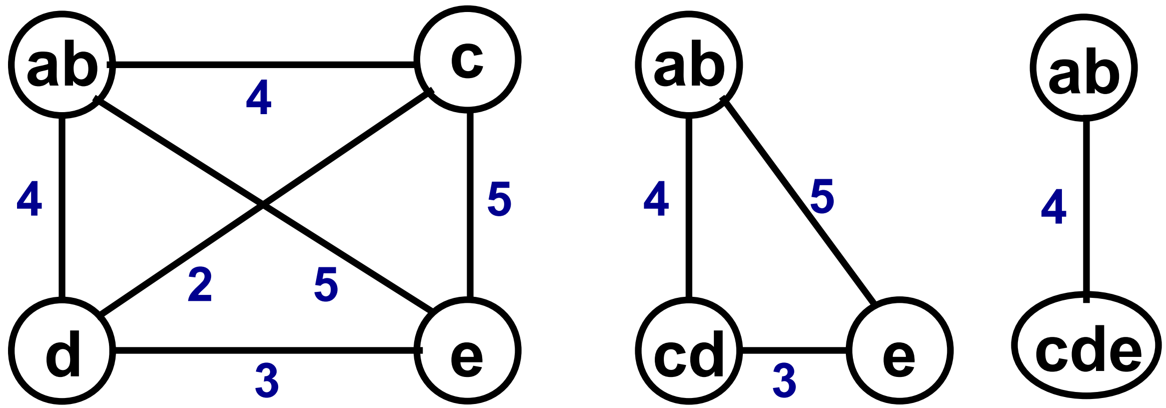
from http://en.wikipedia.org/wiki/Dendrogram

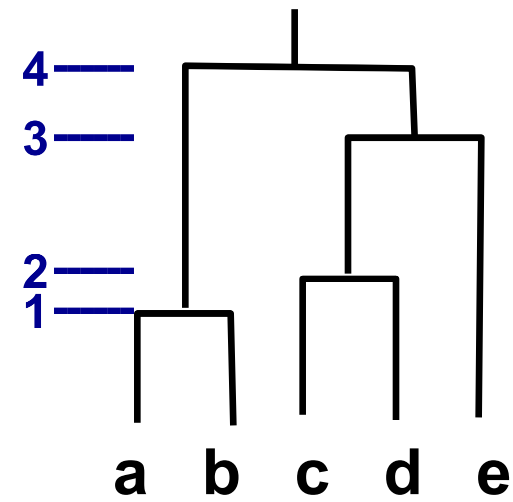
An example HCS
(hierarchical clustering scheme)



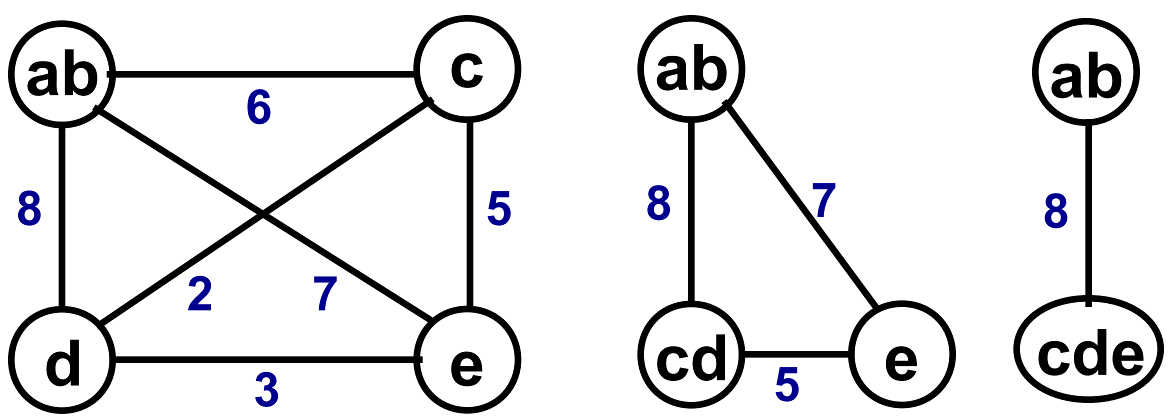
An example HCS
(hierarchical clustering scheme)

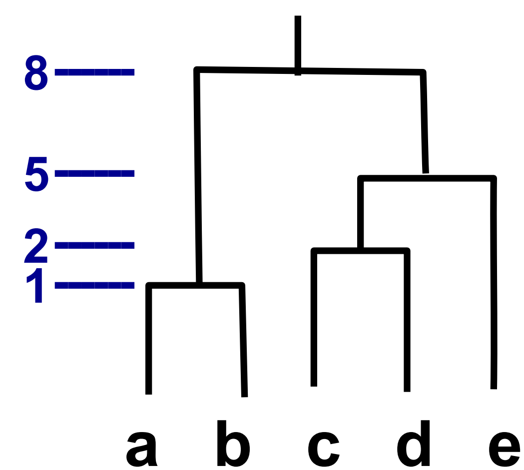
min





max





Region growing

- Start with (random) seed pixel as cluster
- Repeat:
  - Aggregate neighboring pixels that are similar to cluster model
  - Update cluster model with newly incorporated pixels
- This is a generalized floodfill
- When cluster stops growing, begin with new seed pixel and continue
- An easy cluster model:
  - Store mean and covariance of pixels in cluster
  - Use Mahalanobis distance to cluster
    This leads to a natural threshold, e.g., $\pm$ 2.5 σ
  - Update mean and covariance efficiently by keeping track of sum(x) and sum(x^2)
- One danger: Since multiple regions are not grown simultaneously, threshold must be appropriate, or else early regions will dominate

Region growing

```
GrowSingleRegion(I, O, p, label)
model.Initialize( I(p) )
frontier.push(p)
O(p) ← label
while not frontier.isEmpty() do
        p ← frontier.pop()
        for q ∈ N(p) do
                if model.isSimilar( I(q) )
                then frontier.push(q)
                        O(q) ← label
                        model.update( I(q) )
```

```
RegionGrow(I)
label ← 0
for (x, y) ∈ I do
        L(x, y) ← unlabeled
while L(x, y) = unlabeled for some (x, y) do
        p ← GetSeedPixel(I, L)
        L ← GrowSingleRegion(I, L, p, label)
        label ← label + 1
return L
```

Region growing results

Region growing, balloons

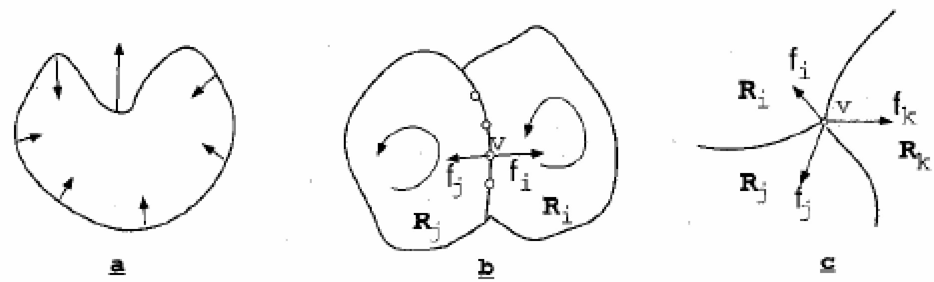
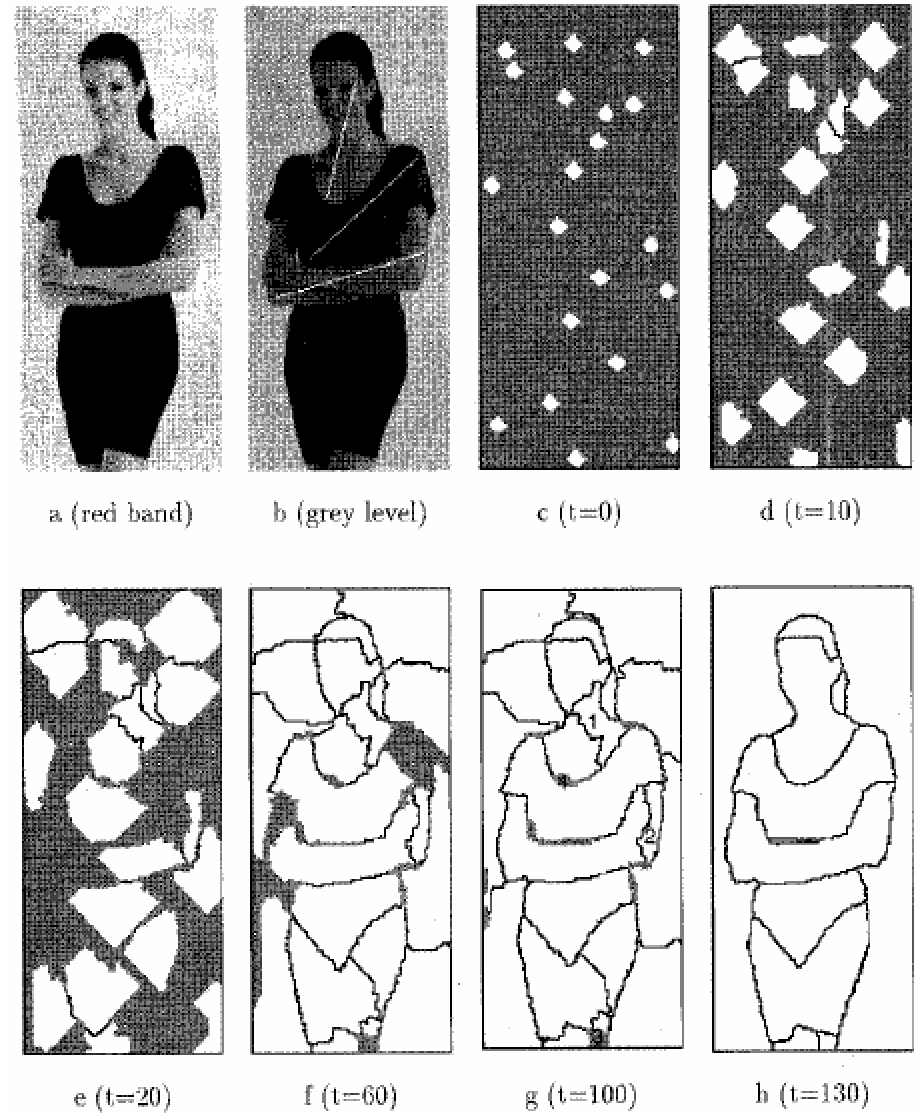


Fig. 2. The forces acting on the contour: (a) the smoothing force, (b the statistics force at a boundary point, (c) the statistics force at a junc tion point.



S. C. Zhu and A.L Yuille**, Region Competition**: Unifying Snake/balloon, Region Growing and Bayes/MDL/Energy for Multi-band Image Segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol.18, no.9, pp.884-900, Sept. 1996.

Stochastic relaxation

- Geman and Geman
- Markov Random Field (MRF)

Minimum spanning tree

- **Agglomerative clustering can be implemented by building graph using pixels as nodes**
- **Repeated merging becomes finding a minimum spanning tree in a graph**

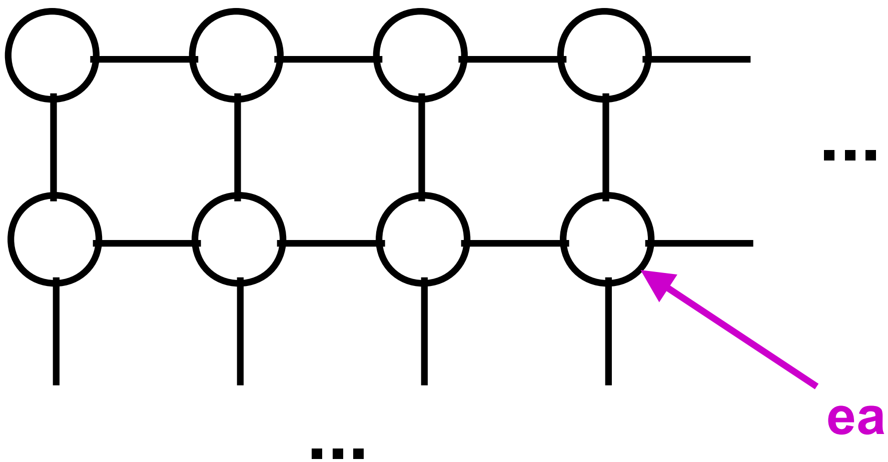
from Pedro F. Felzenszwalb and Daniel P. Huttenlocher, Efficient Graph-Based Image Segmentation, IJCV, 59(2), 2004
http://people.cs.uchicago.edu/~pff/segment/

MST advantages

Minimum spanning tree (MST) answers two important questions unaddressed by region growing:

- How to select the starting pixels?
- Among the several pixels adjacent to the region, which should be considered next?

Image as a graph



Graph G=(V,E)

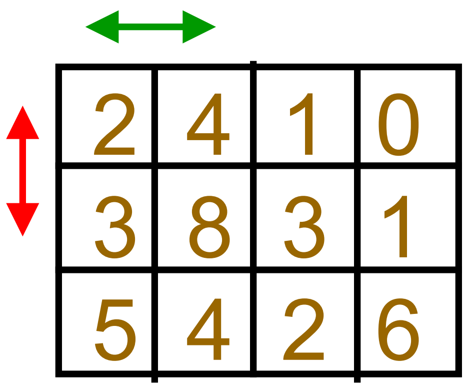
vertices

edges

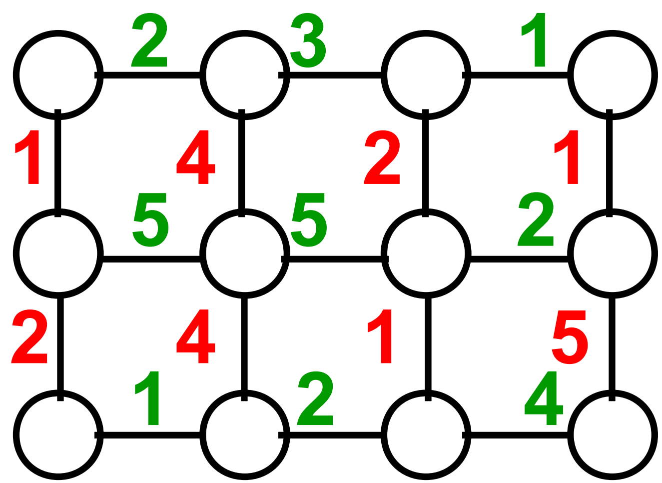
each pixel is a node
in the graph

- vertex v is just a number $v \geq 0$
- edge $e=(u,v)$ is a pair of vertices
- If undirected graph, then $(u,v) \leftrightarrow (v,u)$
- If weighted graph, then $w(e)$ is weight of edge

An example



image



graph

(using absolute difference in intensities)

Minimum spanning tree

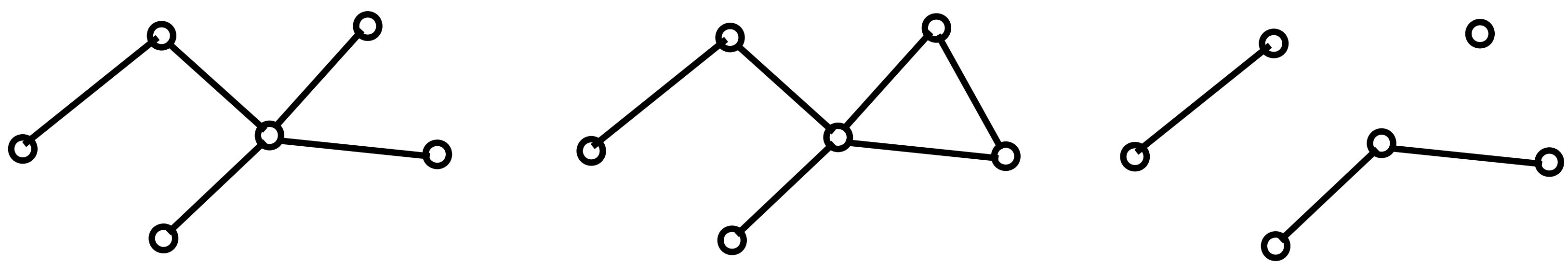
path is sequence of vertices: $v_0, v_1, v_2, \ldots, v_k$

such that (v_i,v_{i+1}) is edge for all i

graph is connected if there exists a path b/w each pair of vertices

graph is tree if connected and acyclic (no cycles)

Examples:



tree | not a tree (contains a cycle) | not a tree (not connected)

Given a graph G=(V,E), the minimum spanning tree is a set of edges such that

- **resulting graph is a tree**
- **the sum of all the edge weights is minimal**

Kruskal's MST algorithm

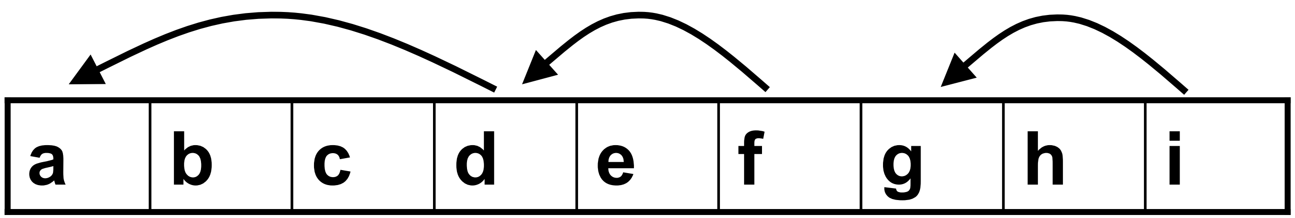
1. Initialize each vertex as separate set (or component or region)
2. Sort edges of E by weight (non-decreasing order)
3. T = ϕ (empty set)
4. for each edge (u,v)
   1. if FindSet(u) $\neq$ FindSet(v)
      1. T = T U {(u,v)}
      2. Merge(u, v)
5. return T

greedy algorithm yet optimal

Kruskal implementation

Kruskal's algorithm is implemented similar to connected components that we saw before

disjoint set data structure (equivalence table):



FindSet(u) recursively traces links (GetEquivalentLabel)
Merge(u,v) simply adds link b/w u and v (SetEquivalence)

How does this relate to image segmentation?
This procedure relates to a single image region; it finds the MST of each region in the image.

Kruskal's MST algorithm

$\textsc{KruskalMST}(I)$

$T \leftarrow \phi$
$\textit{disjoint-set}.\textsc{Initialize}(\textit{width} * \textit{height})$
$E \leftarrow \textsc{ConstructEdges}(I)$
$\langle e_1, \ldots, e_n \rangle \leftarrow \textsc{SortAscendingByWeight}(E)$
$\textbf{for } (u, v) \leftarrow e_1 \textbf{ to } e_n \textbf{ do}$
$\quad \textbf{if } \textit{disjoint-set}.\textsc{FindSet}(u) \neq \textit{disjoint-set}.\textsc{FindSet}(v) \textbf{ then}$
$\quad\quad T \leftarrow T \cup \{(u, v)\}$
$\quad\quad \textit{disjoint-set}.\textsc{Merge}(u, v)$
$\textbf{return } T$

$\textsc{DisjointSet:Initialize}(n)$

$\textbf{for } i \leftarrow 0 \textbf{ to } n - 1 \textbf{ do}$
$\quad \textit{equiv}[i] \leftarrow i$

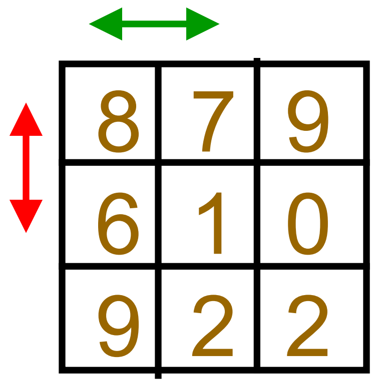
$\textsc{DisjointSet:FindSet}(u)$

$p \leftarrow u$
$\textbf{while } p \neq \textit{equiv}[p] \textbf{ then}$
$\quad p \leftarrow \textit{equiv}[p]$
$\textbf{return } p$

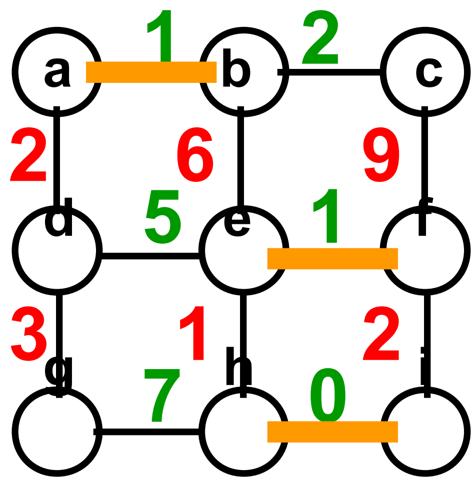
$\textsc{DisjointSet:Merge}(u, v)$

$a \leftarrow \min(\textsc{FindSet}(u), \textsc{FindSet}(v))$
$b \leftarrow \max(\textsc{FindSet}(u), \textsc{FindSet}(v))$
$\textit{equiv}[b] \leftarrow a$

Kruskal example



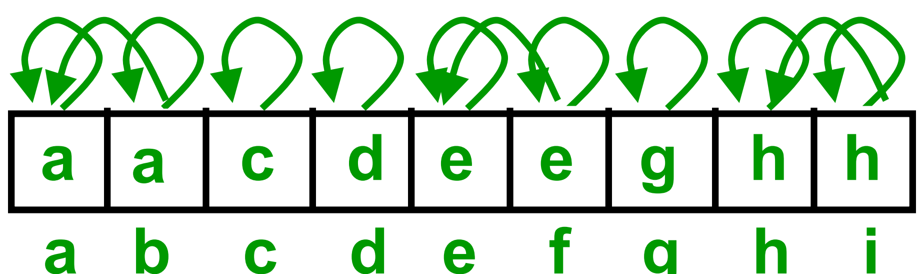
image



graph

Minimum spanning tree

sorted edges: $\langle hi_0, ab_1, ef_1, eh_1, bc_2, fi_2, ad_2, dg_3, de_5, be_6, gh_7, cf_9 \rangle$

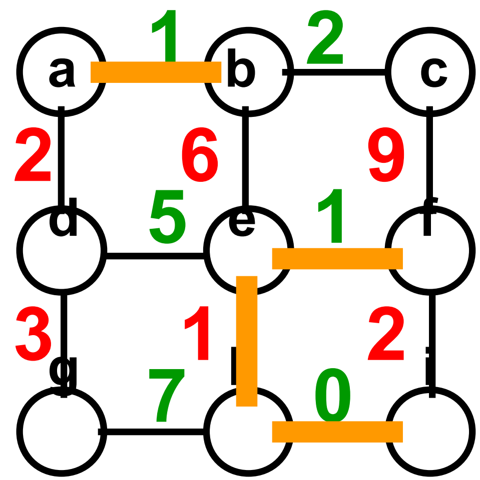


disjoint set data structure

Kruskal example

8	7	9
6	1	0
9	2	2

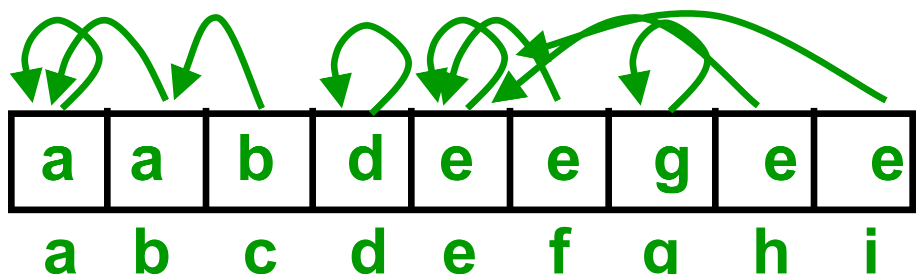
image



Minimum spanning tree (after step 4)

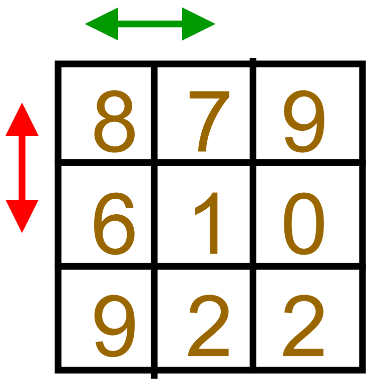
graph

sorted edges: $\langle hi_0, ab_1, ef_1, eh_1, bc_2, fi_2, ad_2, dg_3, de_5, be_6, gh_7, cf_9 \rangle$

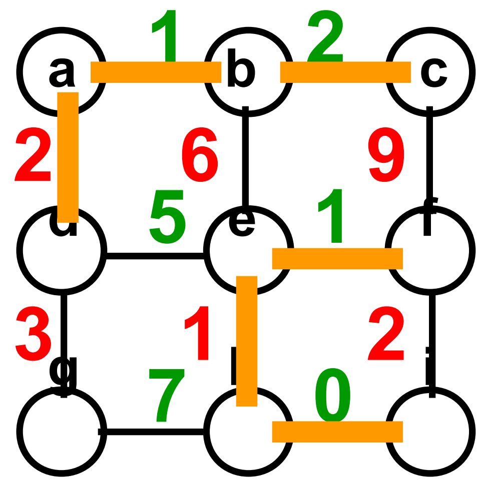


disjoint set data structure

Kruskal example



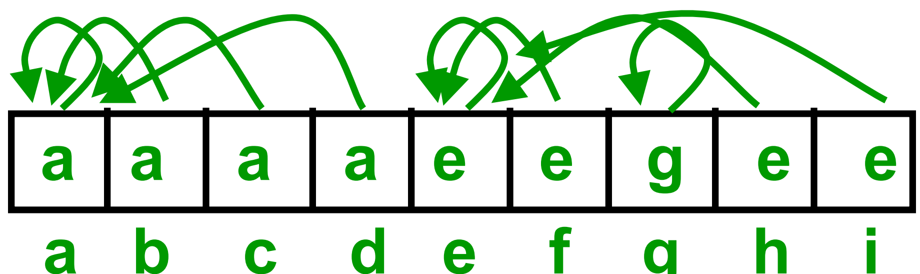
image



Minimum spanning tree (after step 7)

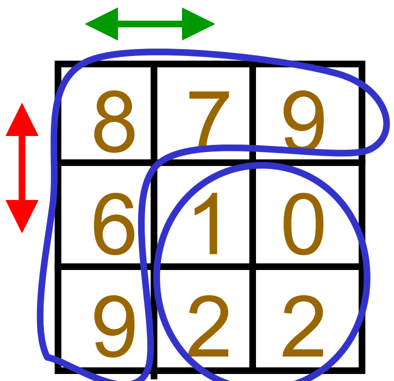
graph

sorted edges: $<hi_0,ab_1,ef_1,eh_1,bc_2,fi_2,ad_2,dg_3,de_5,be_6,gh_7,cf_9>$



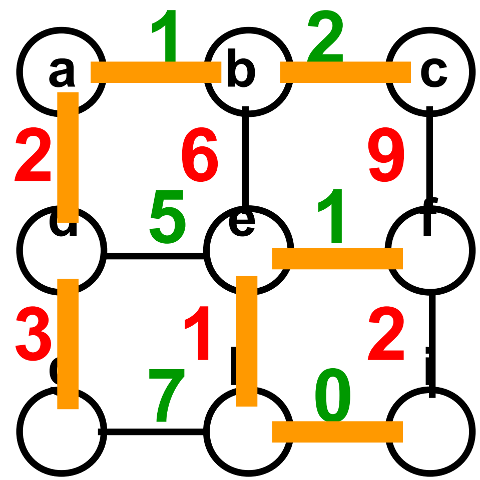
disjoint set data structure

Kruskal example



Note: Pixels grouped correctly!

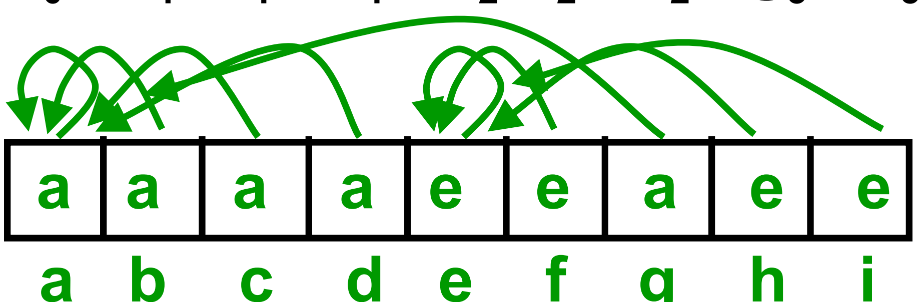
image



Minimum spanning tree (before last step)

graph

sorted edges: $\langle hi_0, ab_1, ef_1, eh_1, bc_2, fi_2, ad_2, dg_3, de_5, be_6, gh_7, cf_9 \rangle$

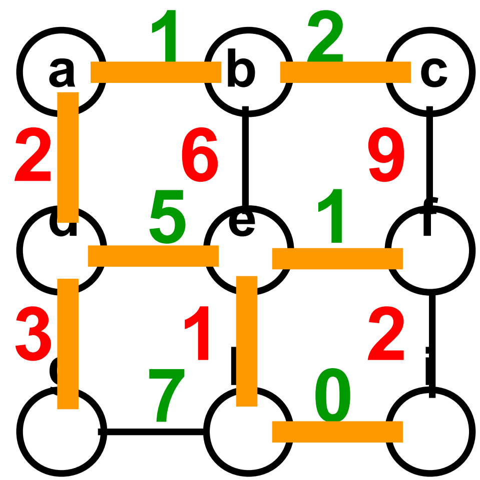


disjoint set data structure

Kruskal example

8	7	9
6	1	0
9	2	2

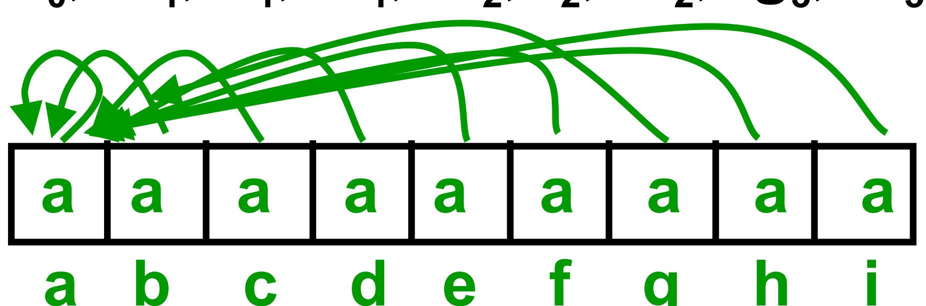
image



Minimum spanning tree (final)

graph

sorted edges: $<hi_0,ab_1,ef_1,eh_1,bc_2,fi_2,ad_2,dg_3,de_5,be_6,gh_7,cf_9>$



disjoint set data structure

MST image segmentation

- First, smooth image by convolving with Gaussian
  - Small variance (e.g., σ^2=0.5) is fine
  - This step is important to produce floating point weights for edges
- Build graph from image:
  - vertices are pixels
  - edges connect adjacent vertices (e.g., 4-adjacency)
  - edge weights are absolute intensity differences:
    $w(u,v) = | I(u) - I(v) |$
- Run Kruskal's algorithm on the graph, but only merge two regions if certain criteria are met
- While running the algorithm,
  - we have a *forest* (set of trees)
  - properties are maintained for each tree
  - trees are merged based on criteria
  - (but we don't care about the trees themselves, so we only need to store the regions, i.e., the set of pixels not edges)
- When the algorithm finishes,
  - the individual trees are the image regions

MST image segmentation

(Felzenszwalb-Huttenlocher IJCV 2004)

- Initialize each pixel as separate set (or component or region)
- Sort edges of *E* by weight (non-decreasing order)
- for each edge (*u*,*v*)
  - if FindSet(*u*) ≠ FindSet(*v*) and

$$w(u,v) < \min(m_u + k/N_u,\ m_v + k/N_v)$$

extra conditions

Note: weight must be floating point, b/c $w(u,v) \geq m_u$ (since edges are considered in non-decreasing order); otherwise, once $k/N_u < 1$, no more merging will occur

- Merge(*u*, *v*)

m_u = maximum weight of all edges in *u* MST
N_u = number of pixels in *u* region
k = parameter (e.g., 150 or 300)

MST image segmentation
(oversimplified version – does not work)

- Initialize each pixel as separate set (or component or region)
- Sort edges of *E* by weight (non-decreasing order)
- for each edge (*u*,*v*)
  - if FindSet(u) ≠ FindSet(v) and

    $$\left.\begin{array}{l}(w(u,v) < m_u \text{ and } w(u,v) < m_v) \\ \text{or } (N_u < k \text{ and } N_v < k)\end{array}\right\}$$ extra conditions

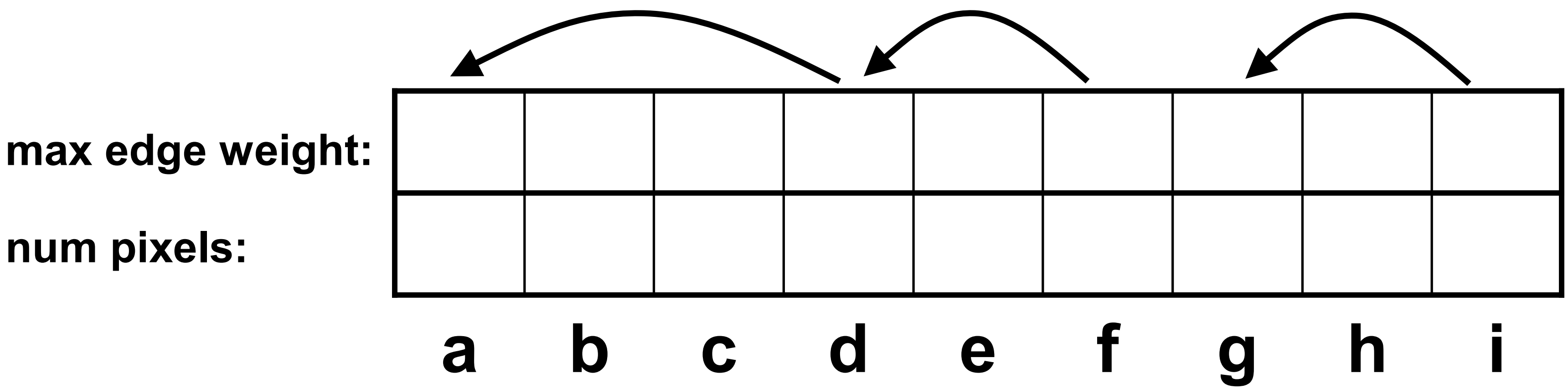
    - Merge(u, v)

m_u = maximum weight of all edges in u MST
N_u = number of pixels in u region
k = parameter (minimum region size)

MST image segmentation

for each region, disjoint set data structure contains:

- max edge weight of all edges in region’s MST
- number of pixels in region



Merge(u,v) is easy:

- set max edge weight to max of two values
- set num pixels to sum of two values

MST segmentation

MST-SEGMENTATION$(I; \sigma, k)$

$I_s \leftarrow$ SMOOTH$(I; \sigma)$ **smoothing necessary for IsSimilar2 to work**
disjoint-set.INITIALIZE$(width * height)$
$E \leftarrow$ CONSTRUCTEDGES(I_s) **weights must be *floats*!**
$\langle e_1, \ldots, e_n \rangle \leftarrow$ SORTASCENDINGBYWEIGHT(E)
for $(u, v) \leftarrow e_1$ **to** e_n **do**
    $u' \leftarrow$ *disjoint-set*.FINDSET(u)
    $v' \leftarrow$ *disjoint-set*.FINDSET(v)
    if $u' \neq v'$ **and** *disjoint-set*.ISSIMILAR2$(w(u, v), u', v'; k)$ **then**
        disjoint-set.MERGE$(u, v, w(u, v))$
for $(x, y) \in I$ **do**
    $L(x, y) \leftarrow$ *disjoint-set*.FINDSET(x, y)
return L

DISJOINTSET:ISSIMILAR2(w, u, v) **do not use integer division!**

return $w < \min(\textit{max-edge-weight}[u] + k / \textit{num-pixels}[u], \textit{max-edge-weight}[v] + k / \textit{num-pixels}[v])$

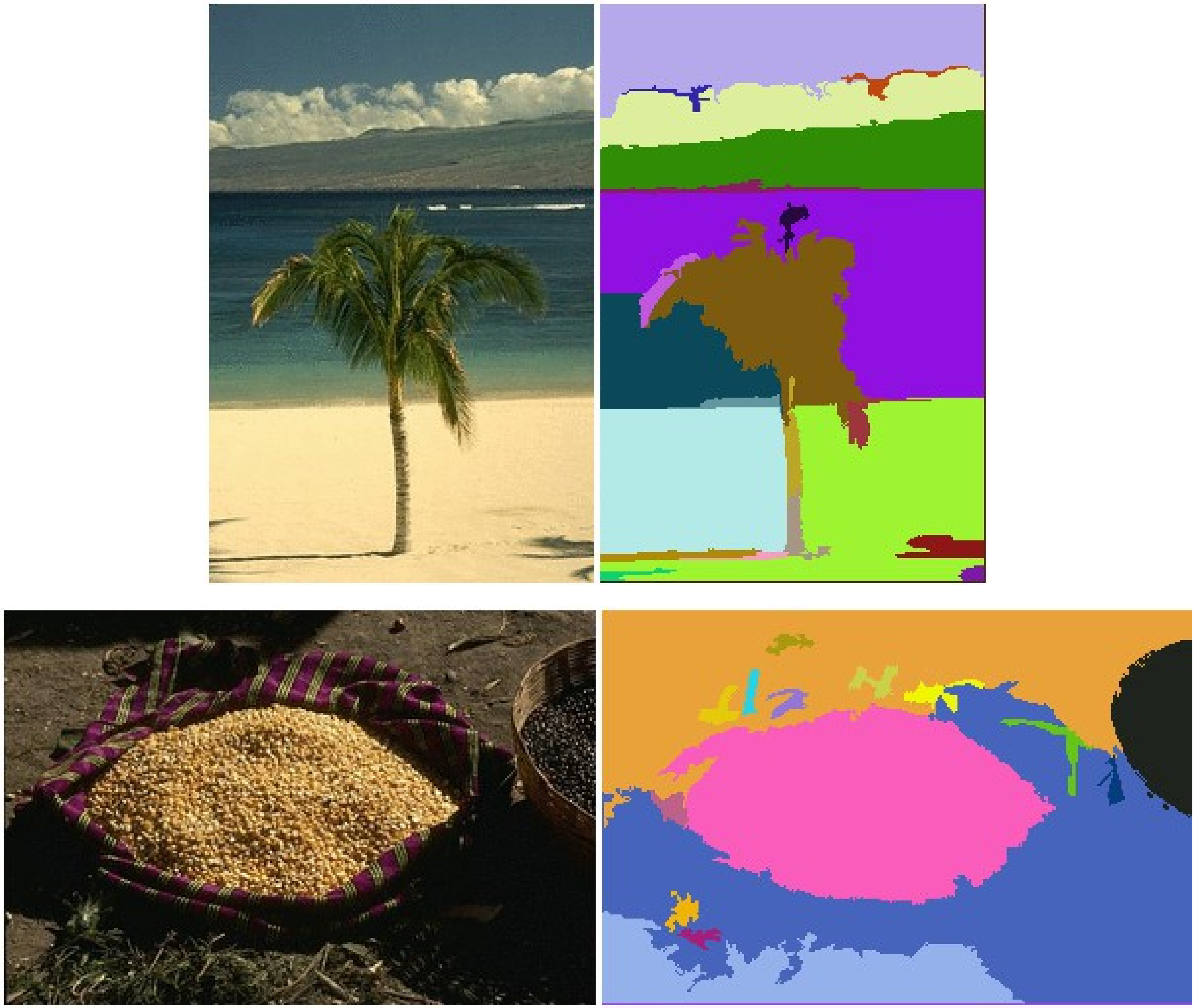
DISJOINTSET:INITIALIZE(n)

for $i \leftarrow 0$ **to** $n - 1$ **do**
    $\textit{equiv}[i] \leftarrow i$
    $\textit{max-edge-weight}[i] \leftarrow 0$
    $\textit{num-pixels}[i] \leftarrow 1$

DISJOINTSET:MERGE(u, v, w)

$a \leftarrow \min(\text{FINDSET}(u), \text{FINDSET}(v))$
$b \leftarrow \max(\text{FINDSET}(u), \text{FINDSET}(v))$ ***(FindSet is redundant if u' v' passed instead)***
$\textit{equiv}[b] \leftarrow a$ ***w* will always be maximum**
$\textit{max-edge-weight}[a] \leftarrow \max(w, \textit{max-edge-weight}[a], \textit{max-edge-weight}[b])$
$\textit{num-pixels}[a] \leftarrow \textit{num-pixels}[a] + \textit{num-pixels}[b]$

Segmentation examples



from Pedro F. Felzenszwalb and Daniel P. Huttenlocher, Efficient Graph-Based Image Segmentation, IJCV, 59(2), 2004
http://people.cs.uchicago.edu/~pff/segment/

More examples

*

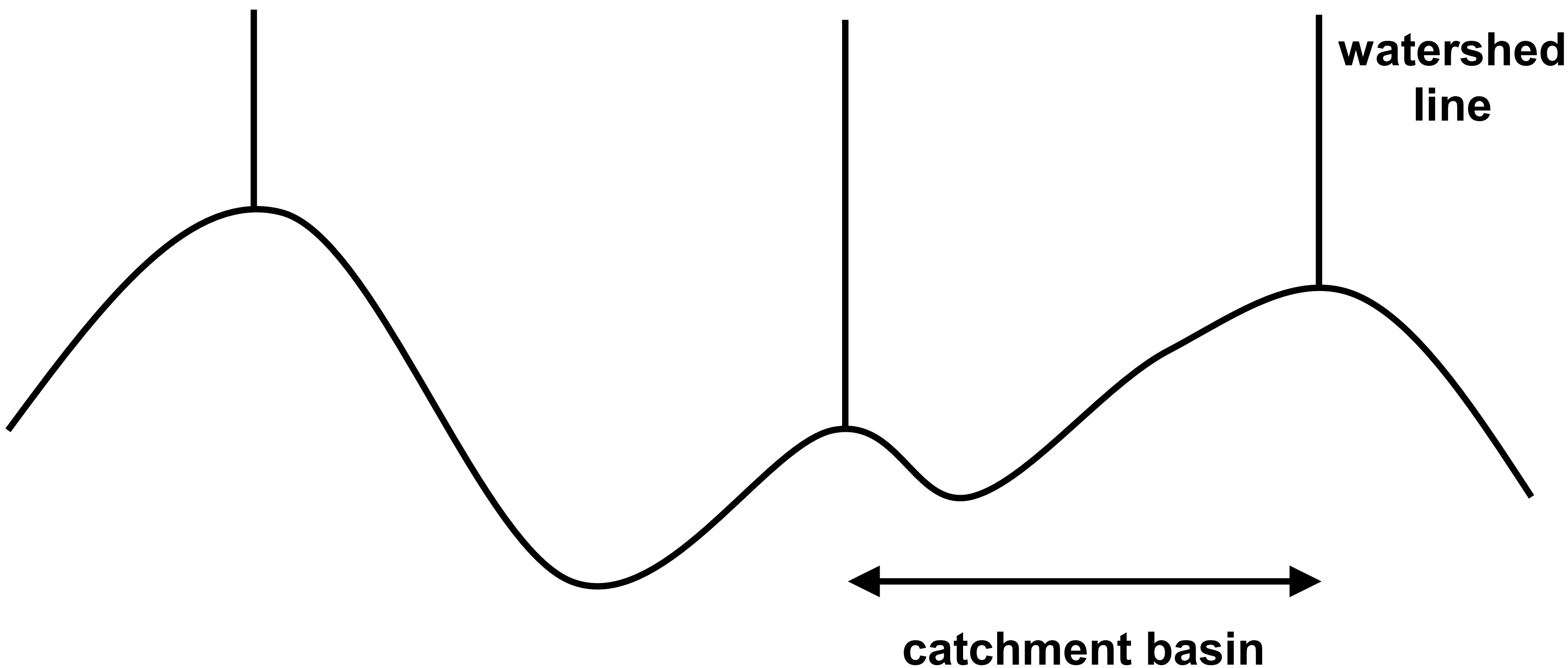
* Pictures from Mean Shift: A Robust Approach toward Feature Space Analysis, by D. Comaniciu and P. Meer http://www.caip.rutgers.edu/~comanici/MSPAMI/msPamiResults.htm

Outline

- Human segmentation
- Standard algorithms:
  - Split-and-merge
  - Region growing
  - Minimum spanning tree
- Watershed algorithm
- Normalized cuts

Watershed segmentation

Interpret (gradient magnitude) image as topographical surface:

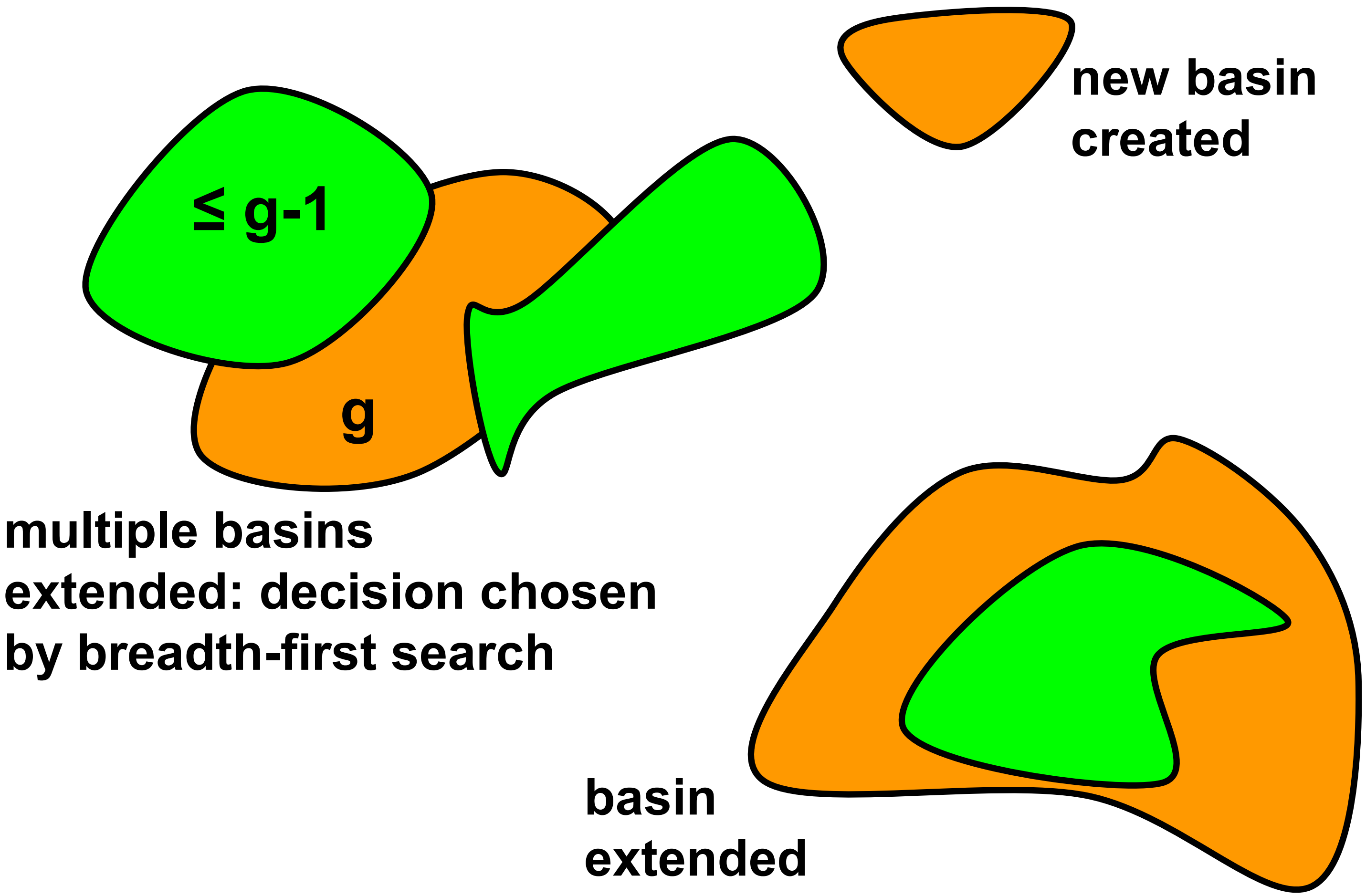


All pixels in catchment basin are connected to minimum by monotonically decreasing path

Watershed algorithms

- Water immersion (Vincent-Soille)
  - Puncture hole at each local minimum, immerse in water
  - Grow level by level, starting with dark pixels
  - *Sorting step:* For efficiency, precompute for each graylevel a list of pixels with that graylevel (histogram with pointers)
  - *Flooding step:* Then, repeat:
    - Breadth-first search (floodfill) of level g given flooding up to level g-1
    - For each pixel with value g, either assign to closest catchment basin or declare new catchment basin (geodesic influence zone)
- Tobogganing
  - Find downstream path from each pixel to local minimum
  - Difficult to define for discrete (quantized) images because of plateaus

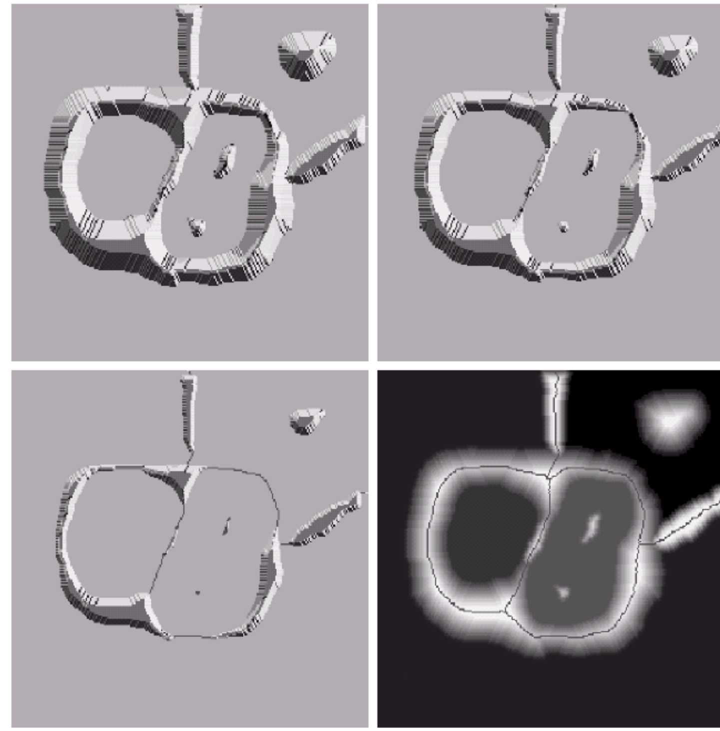
Vincent-Soille algorithm



Watershed

a b
c d

FIGURE 10.44
(a) Original image.
(b) Topographic view. (c)–(d) Two stages of flooding.



e f
g h

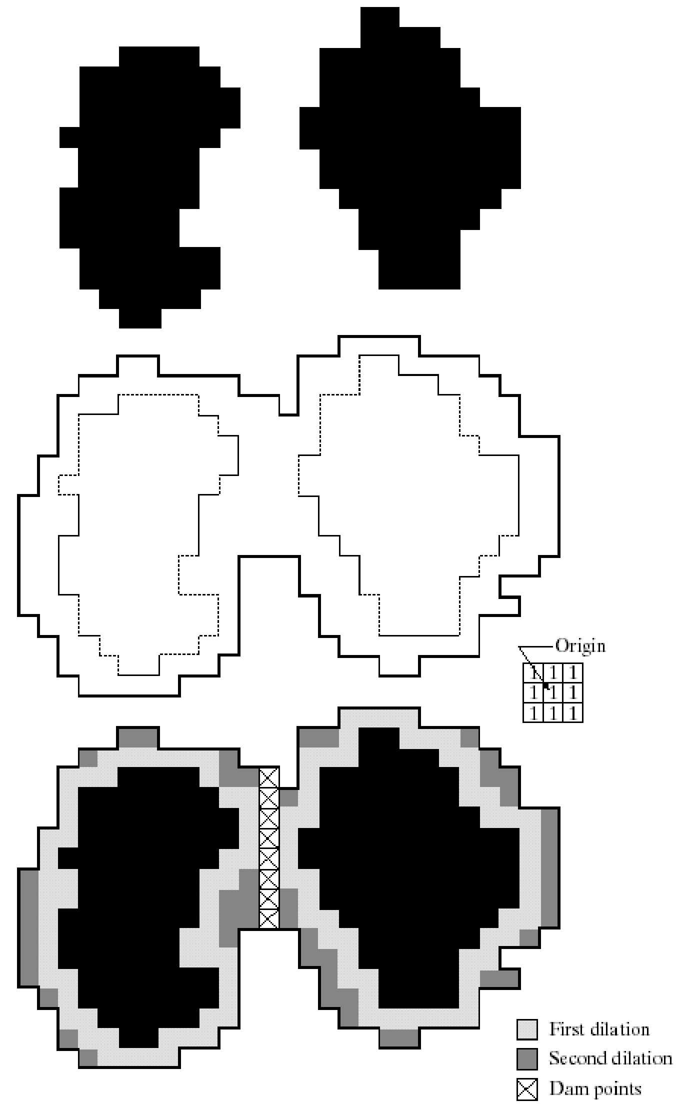
FIGURE 10.44
(Continued)
(e) Result of further flooding.
(f) Beginning of merging of water from two catchment basins (a short dam was built between them). (g) Longer dams. (h) Final watershed (segmentation) lines. (Courtesy of Dr. S. Beucher, CMM/Ecole des Mines de Paris.)

from Gonzalez and Woods, *Digital Image Processing*

Traditional watershed uses dams

a
b c
d

FIGURE 10.45 (a) Two partially flooded catchment basins at stage $n-1$ of flooding. (b) Flooding at stage n, showing that water has spilled between basins (for clarity, water is shown in white rather than black). (c) Structuring element used for dilation. (d) Result of dilation and dam construction.



(But our implementation does not need dams)

Watershed results

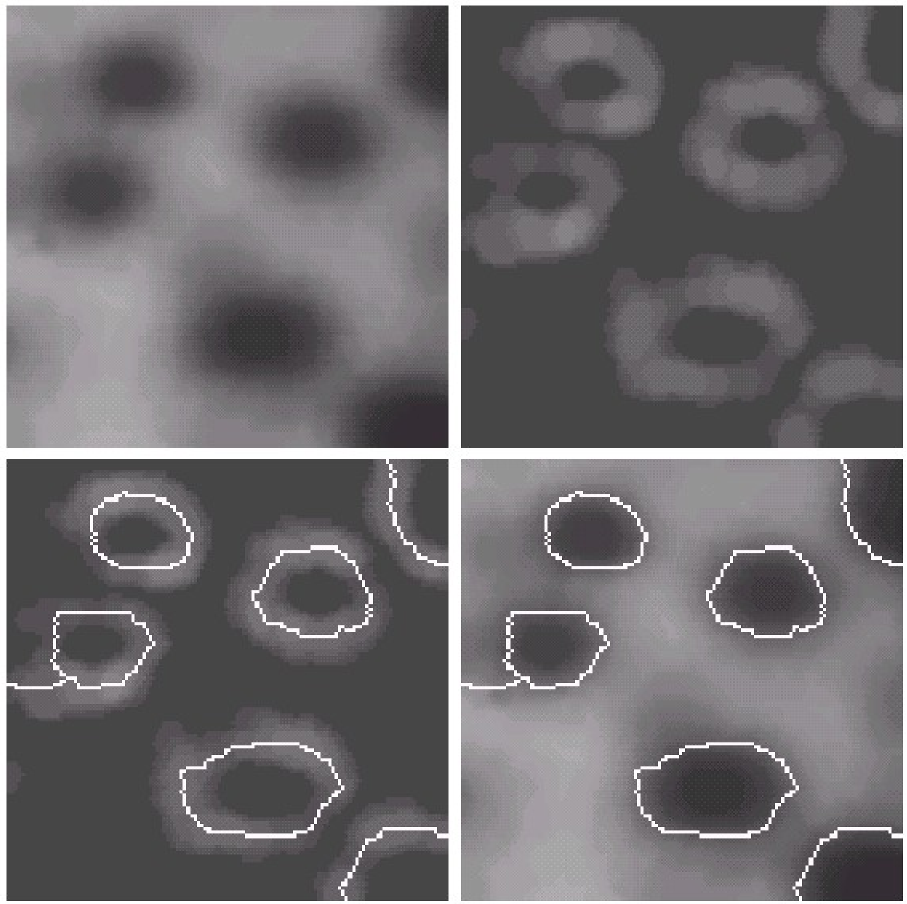
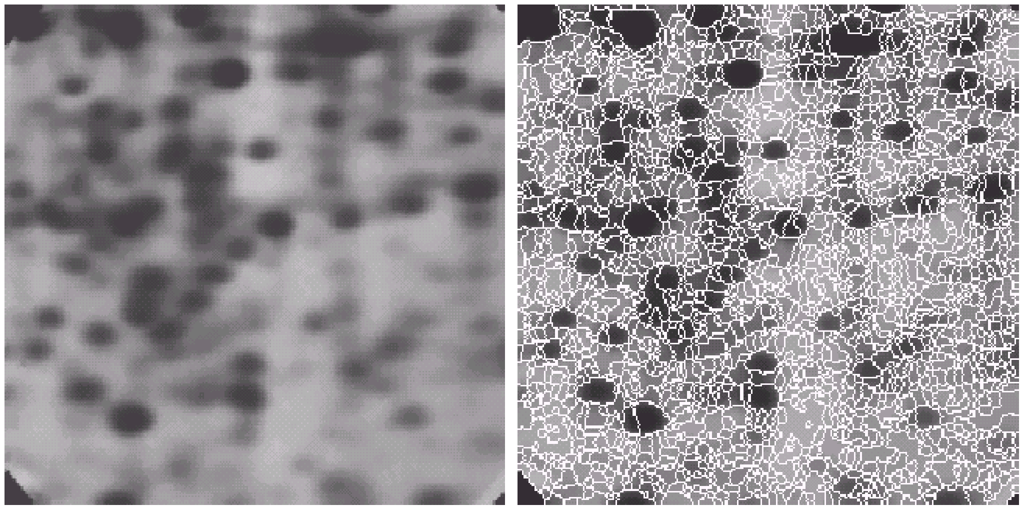


FIGURE 10.46 (a) Image of blobs. (b) Image gradient. (c) Watershed lines. (d) Watershed lines superimposed on original image. (Courtesy of Dr. S. Beucher, CMM/Ecole des Mines de Paris.)

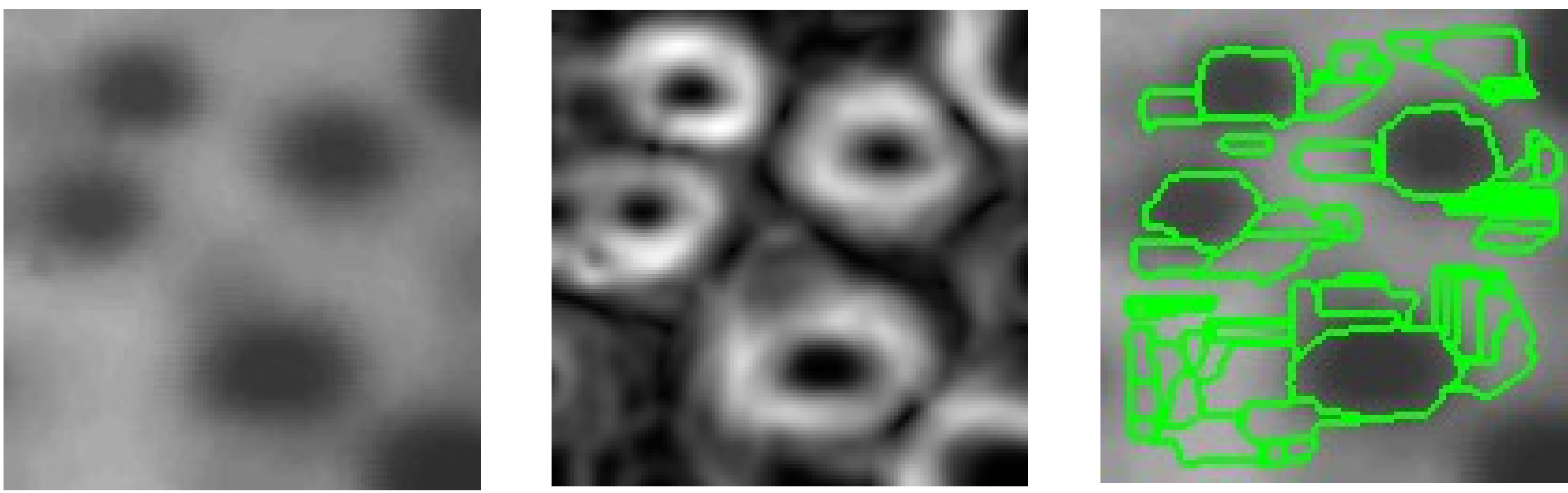
(But the results from our implementation will be better than this)

Watershed leads to oversegmentation

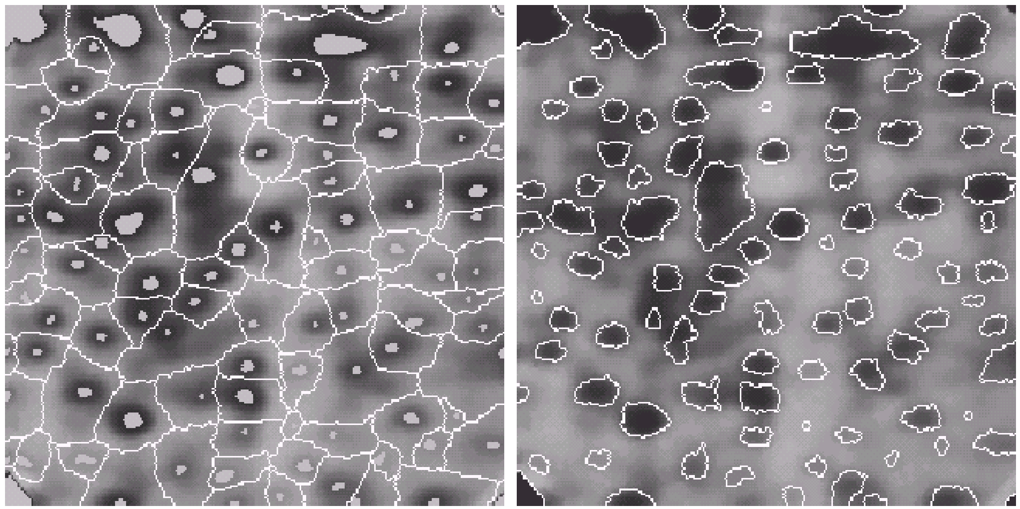


a b

FIGURE 10.47 (a) Electrophoresis image. (b) Result of applying the watershed segmentation algorithm to the gradient image. Oversegmentation is evident. (Courtesy of Dr. S. Beucher, CMM/Ecole des Mines de Paris.)

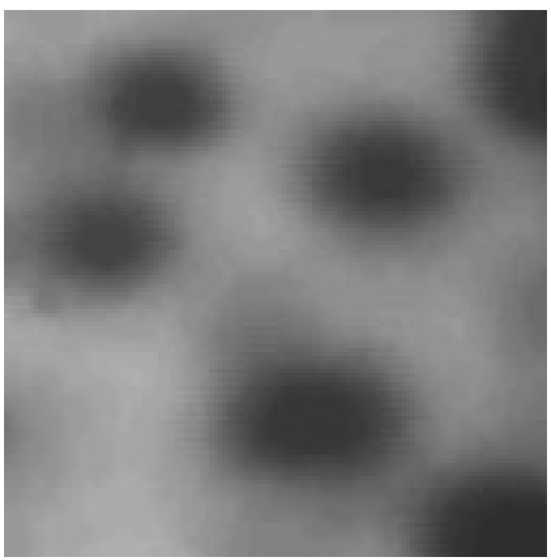


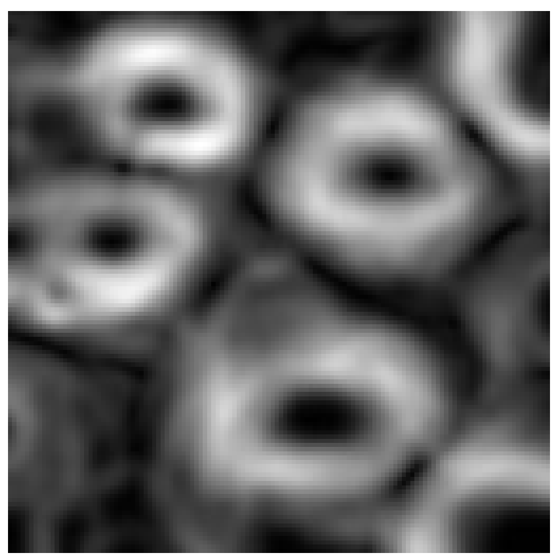
Markers solve this problem

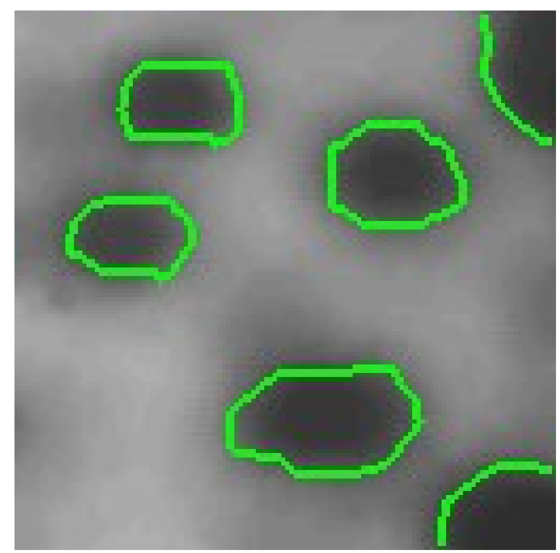


a b

FIGURE 10.48 (a) Image showing internal markers (light gray regions) and external markers (watershed lines). (b) Result of segmentation. Note the improvement over Fig. 10.47(b). (Courtesy of Dr. S. Beucher, CMM/Ecole des Mines de Paris.)







Marker-based watershed

- Threshold image
- Compute chamfer distance
- Run watershed to get lines between objects
- Set these lines (skeletons) and blobs to zero in the gradient magnitude image

  ```
  for each x, y:
     grad(x,y) = max( grad(x,y), 1 )
     if marker(x,y), grad(x,y) = 0
  ```

  (Alternatively, use separate gradient and marker images)
- Only allow new basins where the value is zero

Simplified Vincent-Soilles algorithm (Marker-based)

1. // initalization
   (a) Precompute array of pixel lists for each graylevel g
   (b) Set label[p]=-1 for all pixels p
   (c) globallabel = 0
2. // flood topological surface one graylevel at a time
   for each graylevel g=0 to G,
   (a) // grow existing catchment basins by one pixel, creating initial frontier
   for each pixel p such that img[p]=g and there exists a neighbor q of p for which label[q]≥0 (in an existing catchment basin),
      i. label[p] = label[q]
      ii. `frontier.push_back(p)`
   (b) // continue to grow existing basins one pixel thick each iteration by expanding frontier
   while !frontier.empty(),
      i. `p = frontier.pop_front()`
      ii. for each neighbor q of p such that img[q]==g and label[q]=-1 (unlabeled),
         A. label[q] = label[p]
         B. `frontier.push_back(q)`

   only needs to be done once initially (can use connected components)

   (c) // create new catchment basins
   for each p such that img[p]=g and ~~label[p]=-1 (still unlabeled),~~ marker[p] = true
      i. floodfill region containing p, assigning label 'globallabel++' (floodfill using marker image)

(FIFO queue is important – use std::queue or std::deque)

Simplified Vincent-Soilles algorithm (in detail)
(Marker-based)

1. **// initalization**

   (a) for each graylevel g=0 to G-1,
   pixellist[g].clear()

   (b) for each pixel p in input image f,

   i. pixellist[f(p)].push_back(p) // precompute pixel lists

   ii. label(p) = -1 // all pixels are initially unlabeled

   (c) next_label = 0

   (d) frontier.clear()

2. **// flood topological surface one graylevel at a time**
   for each graylevel g=0 to G-1,

   (a) **// grow existing catchment basins by one pixel, creating initial frontier**
   for each pixel p in pixellist[g],
   for each neighbor q of p,
   if label(q) $\geq$ 0 (in an existing catchment basin),

   i. label(p) = label(q)

   ii. `frontier.push_back(p)`

   (b) **// continue to grow existing basins one pixel thick each iteration by expanding frontier**
   while !frontier.empty(),

   i. `p = frontier.pop_front()`

   ii. for each neighbor q of p
   if f(q) == g and label(q) < 0 (unlabeled),

   A. label(q) = label(p)

   B. `frontier.push_back(q)`

   (c) **// create new catchment basins**
   for each pixel p in pixellist[g],
   if ~~label(p) < 0 (still unlabeled),~~ marker(p) = true

   i. floodfill region containing p, assigning label 'next_label' (floodfill using marker image)

   ii. next_label = next_label + 1

only needs to be done once initially (can use connected components)

FIFO queue is important to ensure that regions grow at equal rates (breadth-first search) (use std::queue or std::deque)

Simplified Vincent-Soilles algorithm (in detail)

```
WATERSHED(I)
    ▷ initialization
    f ← GRADIENTMAGNITUDE(I)
    for each value k ← 0 to n_grad − 1 do
        pixellist[k].CLEAR()
    for each pixel p ∈ f do
        pixellist[f(p)].PUSHBACK(p)                          ▷ precompute pixel lists
        L(p) ← UNLABELED                                     ▷ all pixels are initially unlabeled
    next-label ← 0
    frontier.clear()
    ▷ flood topological surface one value at a time
    for each value k ← 0 to n_grad − 1 do
        ▷ grow existing catchment basins by one pixel, creating initial frontier
        for each pixel p in pixellist[k] do
            if there exists a neighbor q of p such that L(q) ≠ UNLABELED
            then L(p) ← L(q)                                 ▷ (in an existing catchment basin)
                frontier.PUSHBACK(p)
        ▷ continue to grow existing basins one pixel thick each iteration by expanding frontier
        while NOT frontier.EMPTY() do
            p ← frontier.POPFRONT()
            if there exists a neighbor q of p such that f(q) ≤ k and L(q) == UNLABELED
            then L(q) ← L(p)                                 ▷ (unlabeled)
                frontier.PUSHBACK(q)
        ▷ create new catchment basins
        for each pixel p in pixellist[k] do
            if L(p) == UNLABELED                             ▷ (still unlabeled)
            then floodfill region containing p, assigning label next-label
                next-label ← next-label + 1
    return L
```

Watershed example

8	7	5	4	3
9	6	2	1	2
4	2	7	3	4
3	0	1	8	2
6	1	3	6	1

gradmag image

Watershed example

8	7	5	4	3
9	6	2	1	2
4	2	7	3	4
3	0	1	8	2
6	1	3	6	1

gradmag image

0: (1,3)
1:
2:
3:
4:
5:
6:
7:
8:
9:

pixel list

Step 1:
Compute pixel list

Watershed example

8	7	5	4	3
9	6	2	1	2
4	2	7	3	4
3	0	1	8	2
6	1	3	6	1

gradmag image

0: (1,3)
1: (3,1), (2,3), (1,4), (4,4)
2:
3:
4:
5:
6:
7:
8:
9:

pixel list

Step 1:
Compute pixel list

Watershed example

8	7	5	4	3
9	6	2	1	2
4	2	7	3	4
3	0	1	8	2
6	1	3	6	1

gradmag image

0: (1,3)
1: (3,1), (2,3), (1,4), (4,4)
2: (2,1), (4,1), (1,2), (4,3)
3:
4:
5:
6:
7:
8:
9:

pixel list

Step 1:
Compute pixel list

Watershed example

8	7	5	4	3
9	6	2	1	2
4	2	7	3	4
3	0	1	8	2
6	1	3	6	1

gradmag image

0: (1,3)
1: (3,1), (2,3), (1,4), (4,4)
2: (2,1), (4,1), (1,2), (4,3)
3: (4,0), (3,2), (0,3), (2,4)
4: (3,0), (0,2), (4,2)
5: (2,0)
6: (1,1), (0,4), (3,4)
7: (1,0), (2,2)
8: (0,0), (3,3)
9: (0,1)

pixel list

Step 1:
Compute pixel list

Watershed example

8	7	5	4	3
9	6	2	1	2
4	2	7	3	4
3	0	1	8	2
6	1	3	6	1

gradmag image

u	u	u	u	u
u	u	u	u	u
u	u	u	u	u
u	u	u	u	u
u	u	u	u	u

labels

0: (1,3)
1: (3,1), (2,3), (1,4), (4,4)
2: (2,1), (4,1), (1,2), (4,3)
3: (4,0), (3,2), (0,3), (2,4)
4: (3,0), (0,2), (4,2)
5: (2,0)
6: (1,1), (0,4), (3,4)
7: (1,0), (2,2)
8: (0,0), (3,3)
9: (0,1)

pixel list

Step 2:
Set all pixels to "unlabeled"

Watershed example

8	7	5	4	3
9	6	2	1	2
4	2	7	3	4
3	0	1	8	2
6	1	3	6	1

gradmag image

u	u	u	u	u
u	u	u	u	u
u	u	u	u	u
u	u	u	u	u
u	u	u	u	u

labels

0: (1,3)
1: (3,1), (2,3), (1,4), (4,4)
2: (2,1), (4,1), (1,2), (4,3)
3: (4,0), (3,2), (0,3), (2,4)
4: (3,0), (0,2), (4,2)
5: (2,0)
6: (1,1), (0,4), (3,4)
7: (1,0), (2,2)
8: (0,0), (3,3)
9: (0,1)

pixel list

Step 3: k=0
Grow catchment basins
(none to grow)

Watershed example

8	7	5	4	3
9	6	2	1	2
4	2	7	3	4
3	0	1	8	2
6	1	3	6	1

gradmag image

u	u	u	u	u
u	u	u	u	u
u	u	u	u	u
u	u	u	u	u
u	u	u	u	u

labels

0: (1,3)
1: (3,1), (2,3), (1,4), (4,4)
2: (2,1), (4,1), (1,2), (4,3)
3: (4,0), (3,2), (0,3), (2,4)
4: (3,0), (0,2), (4,2)
5: (2,0)
6: (1,1), (0,4), (3,4)
7: (1,0), (2,2)
8: (0,0), (3,3)
9: (0,1)

pixel list

Step 4: k=0
Expand frontier
(no frontier yet)

Watershed example

8	7	5	4	3
9	6	2	1	2
4	2	7	3	4
3	0	1	8	2
6	1	3	6	1

gradmag image

u	u	u	u	u
u	u	u	u	u
u	u	u	u	u
u	0	u	u	u
u	u	u	u	u

labels

0: (1,3)
1: (3,1), (2,3), (1,4), (4,4)
2: (2,1), (4,1), (1,2), (4,3)
3: (4,0), (3,2), (0,3), (2,4)
4: (3,0), (0,2), (4,2)
5: (2,0)
6: (1,1), (0,4), (3,4)
7: (1,0), (2,2)
8: (0,0), (3,3)
9: (0,1)

pixel list

Step 5: k=0
Create new catchment basins

Watershed example

8	7	5	4	3
9	6	2	1	2
4	2	7	3	4
3	0	1	8	2
6	1	3	6	1

gradmag image

u	u	u	u	u
u	u	u	u	u
u	u	u	u	u
u	0	0	u	u
u	0	u	u	u

labels

(shaded pixels are on frontier)

0: (1,3)
1: (3,1), (2,3), (1,4), (4,4)
2: (2,1), (4,1), (1,2), (4,3)
3: (4,0), (3,2), (0,3), (2,4)
4: (3,0), (0,2), (4,2)
5: (2,0)
6: (1,1), (0,4), (3,4)
7: (1,0), (2,2)
8: (0,0), (3,3)
9: (0,1)

pixel list

Step 6: k=1
Grow catchment basins

Watershed example

8	7	5	4	3
9	6	2	1	2
4	2	7	3	4
3	0	1	8	2
6	1	3	6	1

gradmag image

u	u	u	u	u
u	u	u	u	u
u	u	u	u	u
u	0	0	u	u
u	0	u	u	u

labels

(shaded pixels are on frontier)

0: (1,3)
1: (3,1), (2,3), (1,4), (4,4)
2: (2,1), (4,1), (1,2), (4,3)
3: (4,0), (3,2), (0,3), (2,4)
4: (3,0), (0,2), (4,2)
5: (2,0)
6: (1,1), (0,4), (3,4)
7: (1,0), (2,2)
8: (0,0), (3,3)
9: (0,1)

pixel list

Step 7: k=1
Expand frontier
(nowhere to go)

Watershed example

8	7	5	4	3
9	6	2	1	2
4	2	7	3	4
3	0	1	8	2
6	1	3	6	1

gradmag image

u	u	u	u	u
u	u	u	1	u
u	u	u	u	u
u	0	0	u	u
u	0	u	u	2

labels

0: (1,3)
1: (3,1), (2,3), (1,4), (4,4)
2: (2,1), (4,1), (1,2), (4,3)
3: (4,0), (3,2), (0,3), (2,4)
4: (3,0), (0,2), (4,2)
5: (2,0)
6: (1,1), (0,4), (3,4)
7: (1,0), (2,2)
8: (0,0), (3,3)
9: (0,1)

pixel list

Step 8: k=1
Create new basins

Watershed example

8	7	5	4	3
9	6	2	1	2
4	2	7	3	4
3	0	1	8	2
6	1	3	6	1

gradmag image

u	u	u	u	u
u	u	1	1	1
u	0	u	u	u
u	0	0	u	2
u	0	u	u	2

labels

(shaded pixels are on frontier)

0: (1,3)
1: (3,1), (2,3), (1,4), (4,4)
2: (2,1), (4,1), (1,2), (4,3)
3: (4,0), (3,2), (0,3), (2,4)
4: (3,0), (0,2), (4,2)
5: (2,0)
6: (1,1), (0,4), (3,4)
7: (1,0), (2,2)
8: (0,0), (3,3)
9: (0,1)

pixel list

Step 9: k=2
Grow basins

Watershed example

8	7	5	4	3
9	6	2	1	2
4	2	7	3	4
3	0	1	8	2
6	1	3	6	1

gradmag image

u	u	u	u	u
u	u	1	1	1
u	0	u	u	u
u	0	0	u	2
u	0	u	u	2

labels

(shaded pixels are on frontier)

0: (1,3)
1: (3,1), (2,3), (1,4), (4,4)
2: (2,1), (4,1), (1,2), (4,3)
3: (4,0), (3,2), (0,3), (2,4)
4: (3,0), (0,2), (4,2)
5: (2,0)
6: (1,1), (0,4), (3,4)
7: (1,0), (2,2)
8: (0,0), (3,3)
9: (0,1)

pixel list

Step 10: k=2
Expand frontier (nowhere to go)

Watershed example

8	7	5	4	3
9	6	2	1	2
4	2	7	3	4
3	0	1	8	2
6	1	3	6	1

gradmag image

u	u	u	u	u
u	u	1	1	1
u	0	u	u	u
u	0	0	u	2
u	0	u	u	2

labels

0: (1,3)
1: (3,1), (2,3), (1,4), (4,4)
2: (2,1), (4,1), (1,2), (4,3)
3: (4,0), (3,2), (0,3), (2,4)
4: (3,0), (0,2), (4,2)
5: (2,0)
6: (1,1), (0,4), (3,4)
7: (1,0), (2,2)
8: (0,0), (3,3)
9: (0,1)

pixel list

Step 11: k=2
Create new basins
(none to create)

Fast forward…

Watershed example

8	7	5	4	3
9	6	2	1	2
4	2	7	3	4
3	0	1	8	2
6	1	3	6	1

gradmag image

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

labels

0: (1,3)
1: (3,1), (2,3), (1,4), (4,4)
2: (2,1), (4,1), (1,2), (4,3)
3: (4,0), (3,2), (0,3), (2,4)
4: (3,0), (0,2), (4,2)
5: (2,0)
6: (1,1), (0,4), (3,4)
7: (1,0), (2,2)
8: (0,0), (3,3)
9: (0,1)

pixel list

Final result (but note that ties can be broken in other ways)

Marker-based watershed example

8	7	5	4	3
9	6	2	1	2
4	2	7	3	$\cancel{4}$
3	$\cancel{0}$	1	8	2
6	1	3	6	1

gradmag image

There are two markers here, indicated by

$\times$

Marker-based watershed example

8	7	5	4	3
9	6	2	1	2
4	2	7	3	~~4~~
3	~~0~~	1	8	2
6	1	3	6	1

gradmag image

u	u	u	u	u
u	u	u	u	u
u	u	u	u	u
u	u	u	u	u
u	u	u	u	u

labels

0: (1,3)
1: (3,1), (2,3), (1,4), (4,4)
2: (2,1), (4,1), (1,2), (4,3)
3: (4,0), (3,2), (0,3), (2,4)
4: (3,0), (0,2), (4,2)
5: (2,0)
6: (1,1), (0,4), (3,4)
7: (1,0), (2,2)
8: (0,0), (3,3)
9: (0,1)

pixel list

Steps 1 and 2 (initialization) are the same as before

Marker-based watershed example

8	7	5	4	3
9	6	2	1	2
4	2	7	3	4
3	0	1	8	2
6	1	3	6	1

gradmag image

u	u	u	u	u
u	u	u	u	u
u	u	u	u	u
u	u	u	u	u
u	u	u	u	u

labels

0: (1,3)
1: (3,1), (2,3), (1,4), (4,4)
2: (2,1), (4,1), (1,2), (4,3)
3: (4,0), (3,2), (0,3), (2,4)
4: (3,0), (0,2), (4,2)
5: (2,0)
6: (1,1), (0,4), (3,4)
7: (1,0), (2,2)
8: (0,0), (3,3)
9: (0,1)

pixel list

Step 3: k=0
Grow catchment basins
(none to grow)

Marker-based watershed example

8	7	5	4	3
9	6	2	1	2
4	2	7	3	$\cancel{4}$
3	$\cancel{0}$	1	8	2
6	1	3	6	1

gradmag image

u	u	u	u	u
u	u	u	u	u
u	u	u	u	u
u	u	u	u	u
u	u	u	u	u

labels

0: (1,3)
1: (3,1), (2,3), (1,4), (4,4)
2: (2,1), (4,1), (1,2), (4,3)
3: (4,0), (3,2), (0,3), (2,4)
4: (3,0), (0,2), (4,2)
5: (2,0)
6: (1,1), (0,4), (3,4)
7: (1,0), (2,2)
8: (0,0), (3,3)
9: (0,1)

pixel list

Step 4: k=0
Expand frontier
(no frontier yet)

Marker-based watershed example

8	7	5	4	3
9	6	2	1	2
4	2	7	3	$\cancel{4}$
3	$\cancel{0}$	1	8	2
6	1	3	6	1

gradmag image

u	u	u	u	u
u	u	u	u	u
u	u	u	u	1
u	0	u	u	u
u	u	u	u	u

labels

0: (1,3)
1: (3,1), (2,3), (1,4), (4,4)
2: (2,1), (4,1), (1,2), (4,3)
3: (4,0), (3,2), (0,3), (2,4)
4: (3,0), (0,2), (4,2)
5: (2,0)
6: (1,1), (0,4), (3,4)
7: (1,0), (2,2)
8: (0,0), (3,3)
9: (0,1)

pixel list

Step 5: k=0
Create new catchment basins (at markers)

Marker-based watershed example

8	7	5	4	3
9	6	2	1	2
4	2	7	3	~~4~~
3	~~0~~	1	8	2
6	1	3	6	1

gradmag image

u	u	u	u	u
u	u	u	u	u
u	u	u	u	1
u	0	0	u	u
u	0	u	u	u

labels

(shaded pixels are on frontier)

0: (1,3)
1: (3,1), (2,3), (1,4), (4,4)
2: (2,1), (4,1), (1,2), (4,3)
3: (4,0), (3,2), (0,3), (2,4)
4: (3,0), (0,2), (4,2)
5: (2,0)
6: (1,1), (0,4), (3,4)
7: (1,0), (2,2)
8: (0,0), (3,3)
9: (0,1)

pixel list

Step 6: k=1
Grow catchment basins

Marker-based watershed example

8	7	5	4	3
9	6	2	1	2
4	2	7	3	~~4~~
3	~~0~~	1	8	2
6	1	3	6	1

gradmag image

u	u	u	u	u
u	u	u	u	u
u	u	u	u	1
u	0	0	u	u
u	0	u	u	u

labels

(shaded pixels are on frontier)

0: (1,3)
1: (3,1), (2,3), (1,4), (4,4)
2: (2,1), (4,1), (1,2), (4,3)
3: (4,0), (3,2), (0,3), (2,4)
4: (3,0), (0,2), (4,2)
5: (2,0)
6: (1,1), (0,4), (3,4)
7: (1,0), (2,2)
8: (0,0), (3,3)
9: (0,1)

pixel list

Step 7: k=1
Expand frontier
(nowhere to expand)

Marker-based watershed example

8	7	5	4	3
9	6	2	1	2
4	2	7	3	~~4~~
3	~~0~~	1	8	2
6	1	3	6	1

gradmag image

u	u	u	u	u
u	u	u	u	u
u	u	u	u	1
u	0	0	u	u
u	0	u	u	u

labels

0: (1,3)
1: (3,1), (2,3), (1,4), (4,4)
2: (2,1), (4,1), (1,2), (4,3)
3: (4,0), (3,2), (0,3), (2,4)
4: (3,0), (0,2), (4,2)
5: (2,0)
6: (1,1), (0,4), (3,4)
7: (1,0), (2,2)
8: (0,0), (3,3)
9: (0,1)

pixel list

Skip step to create new catchment basins, because this is only done at markers

Marker-based watershed example

8	7	5	4	3
9	6	2	1	2
4	2	7	3	~~4~~
3	~~0~~	1	8	2
6	1	3	6	1

gradmag image

u	u	u	u	u
u	u	u	u	1
u	0	u	u	1
u	0	0	u	1
u	0	u	u	u

labels

(shaded pixels are on frontier)

0: (1,3)
1: (3,1), (2,3), (1,4), (4,4)
2: (2,1), (4,1), (1,2), (4,3)
3: (4,0), (3,2), (0,3), (2,4)
4: (3,0), (0,2), (4,2)
5: (2,0)
6: (1,1), (0,4), (3,4)
7: (1,0), (2,2)
8: (0,0), (3,3)
9: (0,1)

pixel list

Step 8: k=2
Grow catchment basins

Marker-based watershed example

8	7	5	4	3
9	6	2	1	2
4	2	7	3	~~4~~
3	~~0~~	1	8	2
6	1	3	6	1

gradmag image

u	u	u	u	u
u	u	1	1	1
u	0	u	u	1
u	0	0	u	1
u	0	u	u	1

labels

(shaded pixels are on frontier)

0: (1,3)
1: (3,1), (2,3), (1,4), (4,4)
2: (2,1), (4,1), (1,2), (4,3)
3: (4,0), (3,2), (0,3), (2,4)
4: (3,0), (0,2), (4,2)
5: (2,0)
6: (1,1), (0,4), (3,4)
7: (1,0), (2,2)
8: (0,0), (3,3)
9: (0,1)

pixel list

Step 9: k=2
Expand frontier

Marker-based watershed example

8	7	5	4	3
9	6	2	1	2
4	2	7	3	~~4~~
3	~~0~~	1	8	2
6	1	3	6	1

gradmag image

u	u	u	u	1
u	u	1	1	1
u	0	u	1	1
0	0	0	u	1
u	0	0	u	1

labels

(shaded pixels are on frontier)

0: (1,3)
1: (3,1), (2,3), (1,4), (4,4)
2: (2,1), (4,1), (1,2), (4,3)
3: (4,0), (3,2), (0,3), (2,4)
4: (3,0), (0,2), (4,2)
5: (2,0)
6: (1,1), (0,4), (3,4)
7: (1,0), (2,2)
8: (0,0), (3,3)
9: (0,1)

pixel list

Step 10: k=3
Grow catchment basins

Marker-based watershed example

8	7	5	4	3
9	6	2	1	2
4	2	7	3	~~4~~
3	~~0~~	1	8	2
6	1	3	6	1

gradmag image

u	u	u	u	1
u	u	1	1	1
u	0	u	1	1
0	0	0	u	1
u	0	0	u	1

labels

(shaded pixels are on frontier)

0: (1,3)
1: (3,1), (2,3), (1,4), (4,4)
2: (2,1), (4,1), (1,2), (4,3)
3: (4,0), (3,2), (0,3), (2,4)
4: (3,0), (0,2), (4,2)
5: (2,0)
6: (1,1), (0,4), (3,4)
7: (1,0), (2,2)
8: (0,0), (3,3)
9: (0,1)

pixel list

Step 11: k=3
Expand frontier
(nowhere to expand)

Fast forward…

Marker-based watershed example

8	7	5	4	3
9	6	2	1	2
4	2	7	3	~~4~~
3	~~0~~	1	8	2
6	1	3	6	1

gradmag image

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

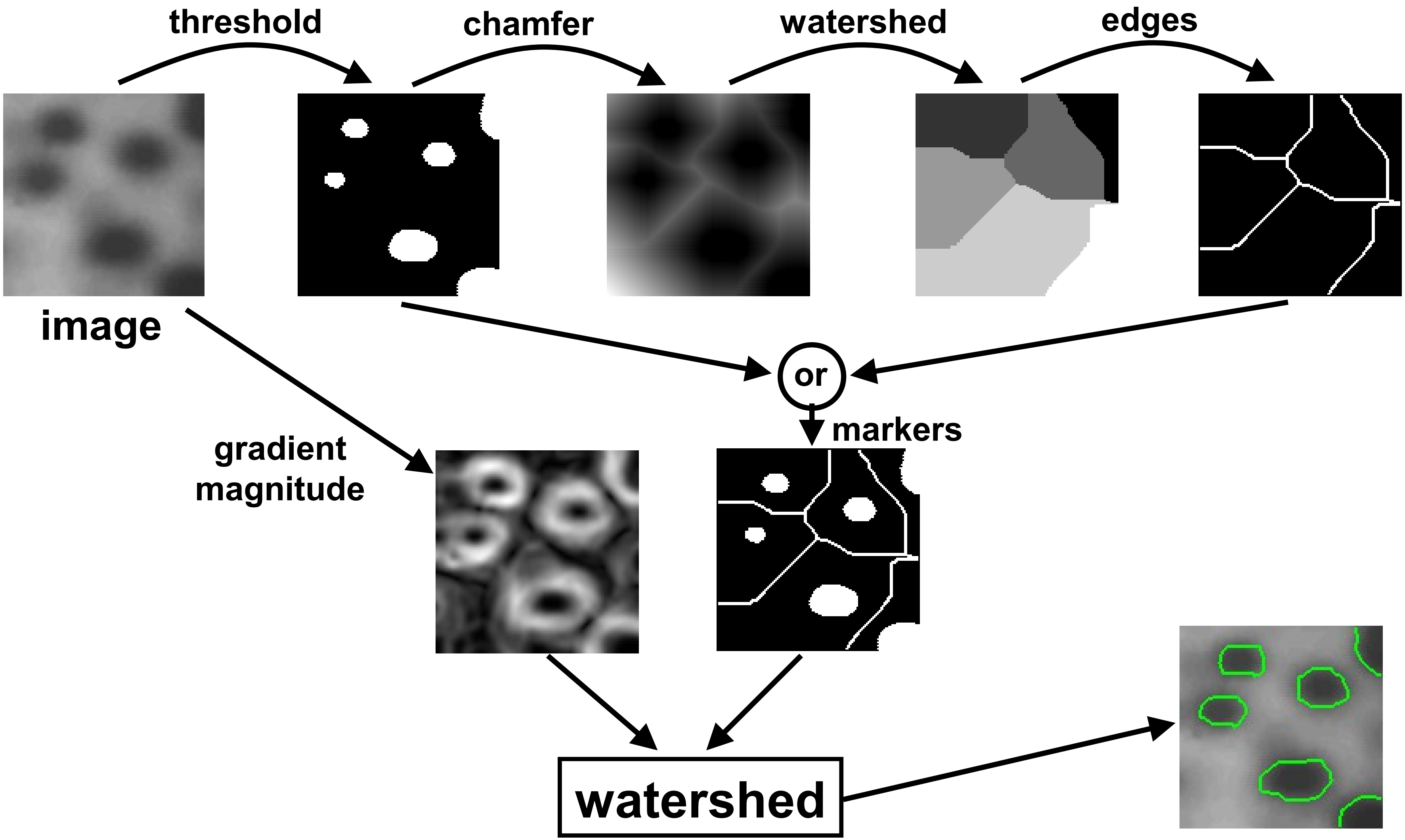
labels

0: (1,3)
1: (3,1), (2,3), (1,4), (4,4)
2: (2,1), (4,1), (1,2), (4,3)
3: (4,0), (3,2), (0,3), (2,4)
4: (3,0), (0,2), (4,2)
5: (2,0)
6: (1,1), (0,4), (3,4)
7: (1,0), (2,2)
8: (0,0), (3,3)
9: (0,1)

pixel list

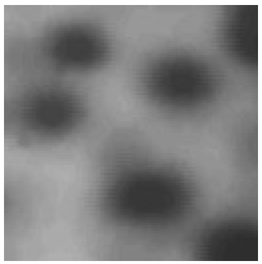
Final result (but note that ties can be broken in other ways)

Marker-based watershed



Why are ridges needed?

threshold

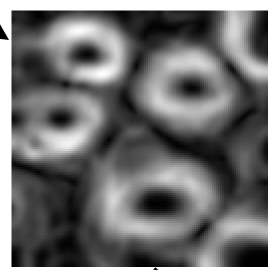


image

• Ridge markers define background region

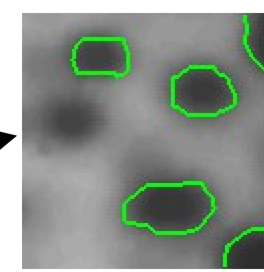
• Without them, one object will spill over into background and disappear

gradient magnitude

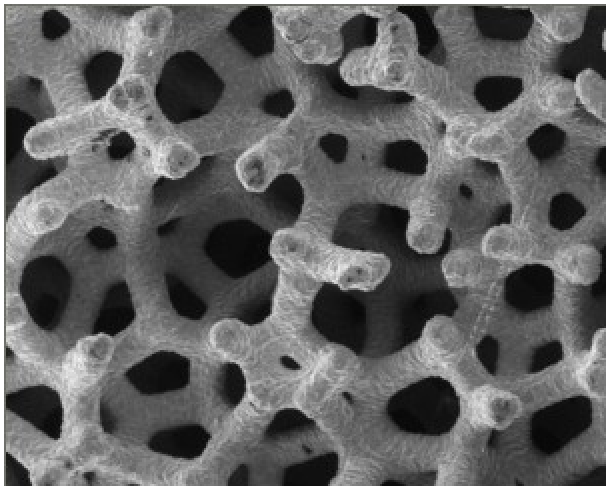


markers

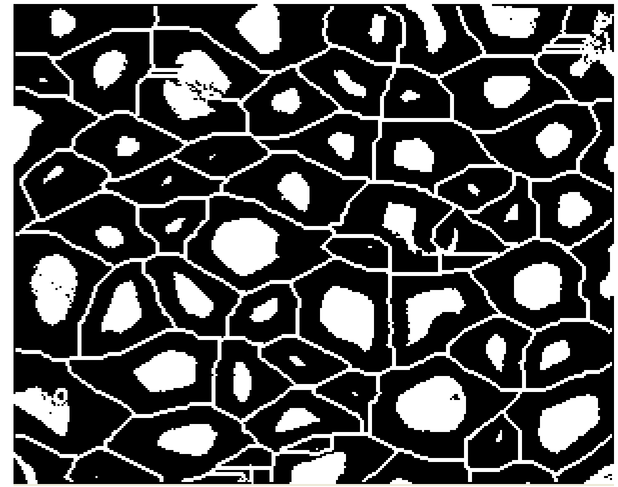
watershed



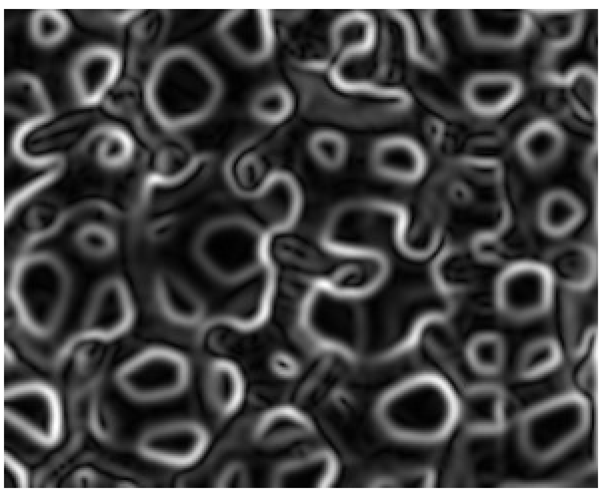
Another example



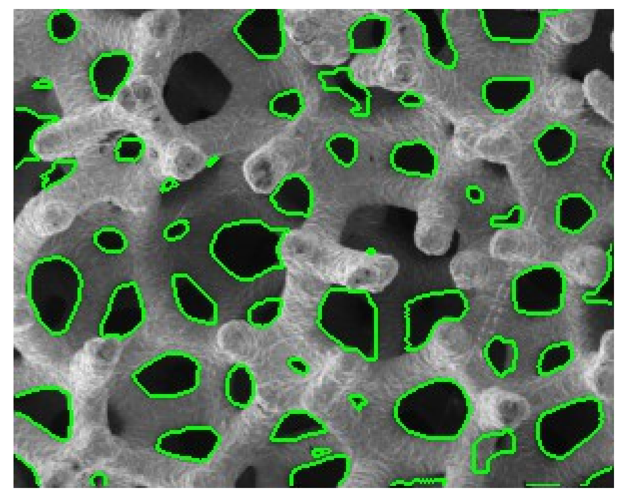
image



markers



gradient magnitude



watershed