Connected Component Tree

Jean Cousty

FOUR-DAY COURSE
on
Mathematical Morphology in image analysis
Bangalore 19-22 October 2010

Motivation

- Connectivity plays important role for image processing
- Morphological connected filters
  - Morphological filters acting on components of a set
  - Morphological filters acting on components of the flat zones of a function
- Graphs are adapted for handling digital connectivity

Problem

- **How can connected filters be efficiently implemented?**

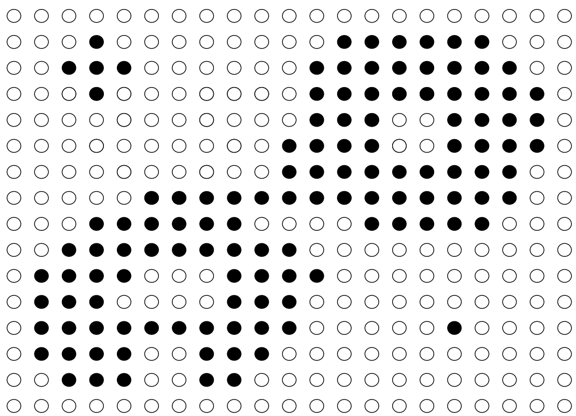
Outline

- Connected filters on (weighted) graphs
  - Connected components
  - Component tree
  - Example of application
- Algorithms for connected filters on (weighted) graphs
  - Union of disjoint set
  - Connected component labeling
  - Component tree extraction

Structuring a discrete set

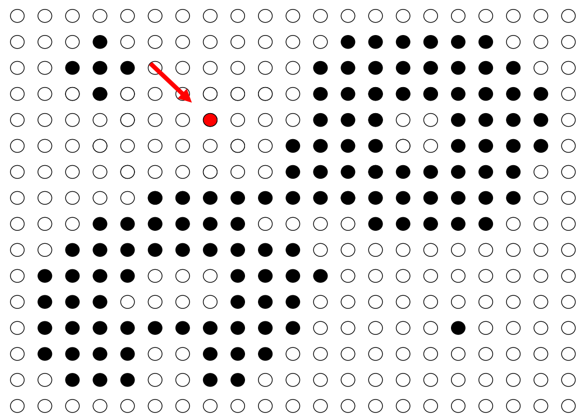
- Let V be a set
  - (*eg*, the set of all image pixels)
- Let X be a subset of V
  - (*eg*, a binary object to be processed)

Discrete set



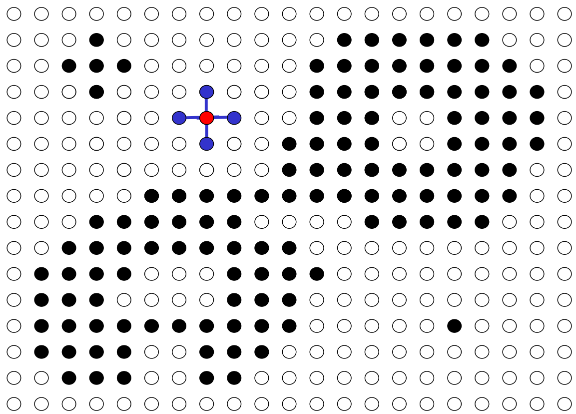
V (white & black dots) and X (black dots)

Discrete set



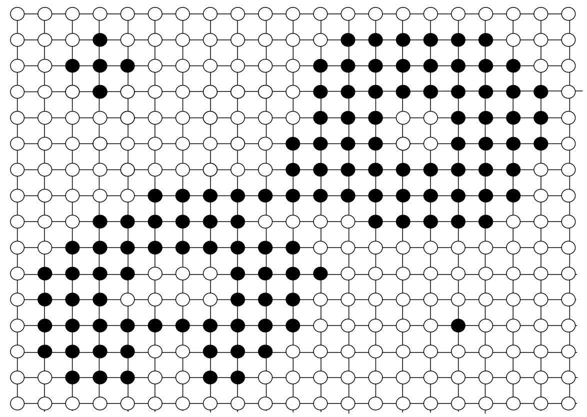
V (white & black dots) and X (black dots)

Graph



Neighborhood (in blue) of a point (in red)

Graph



Vertex set (dots) and edge set E (line segments) of a graph (V,E)

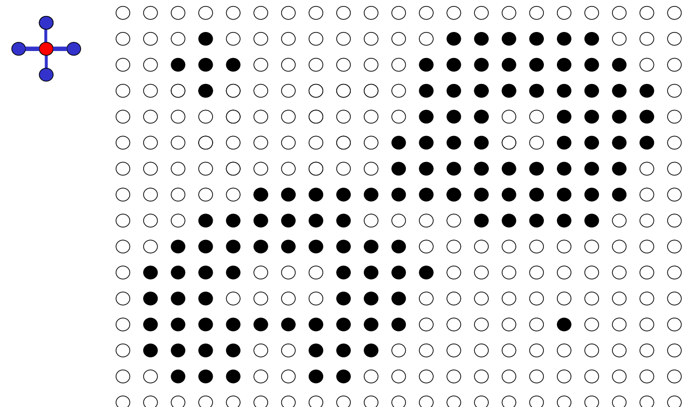
Connected sets

- Let $G = (V,E)$ be a (undirected) graph
- Let X be a subset of V

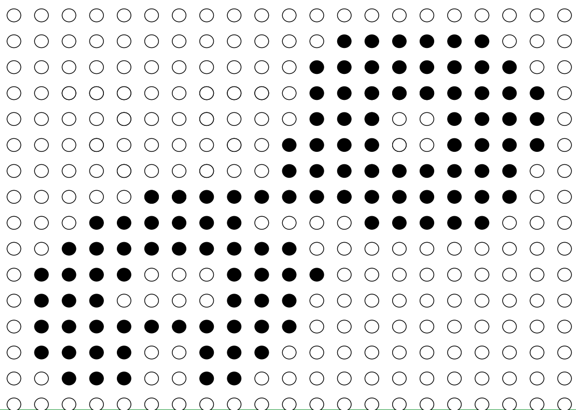
Definition

- X is ***connected*** if
  - there exists a path between any two points of X

Example: Discrete set that is not connected



Example: Connected discrete set



Connected components

- Let $G = (V,E)$ be a (undirected) graph.
- Let X be a subset of V
- Let Y be a subset of X

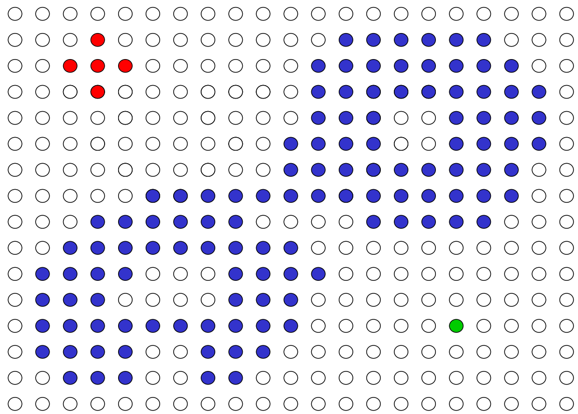
Definition

- *Y is a* **(connected) component of X** *if*
  - *Y is connected*
  - *no proper superset of Y included in X is connected*

Property

- *Y is a* **component of X** *iff for any x in Y*
  - *Y is the union of all connected subsets of X that conains x*

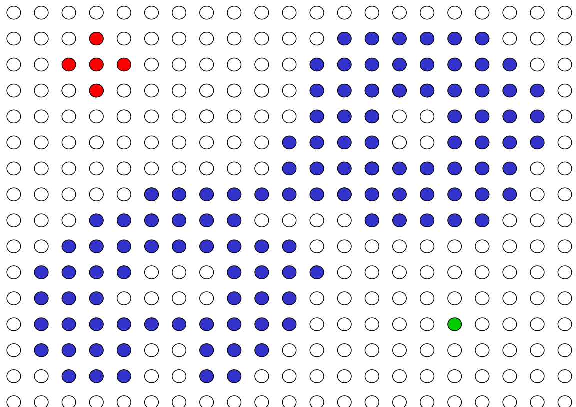
Example : Connected components partition



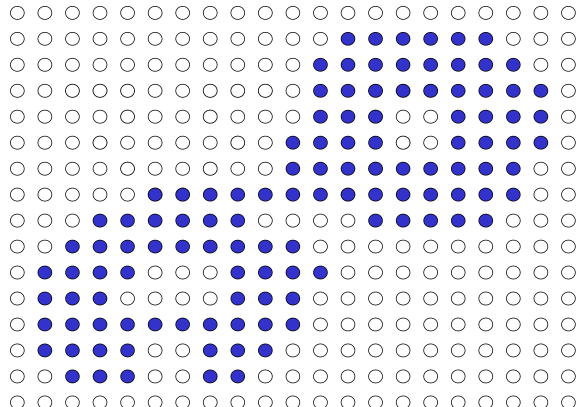
Connected opening a discrete set

- According to a given criterion (*e.g.*, the size), keep the most important components of X

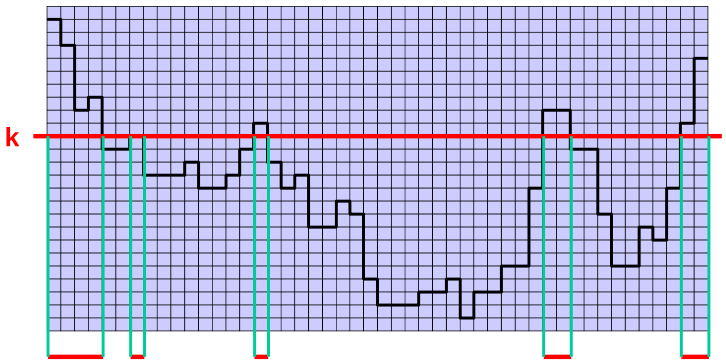
Example: area filtering a discrete set



Example: area filtering a discrete set

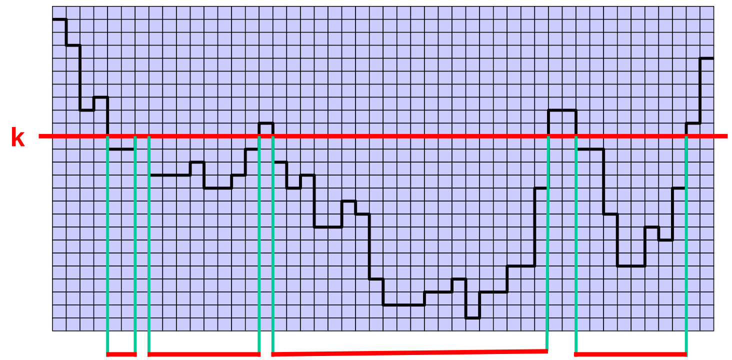


From maps to sets



$F_k = \{x \in E, F(x) \geq k\}$, i.e., points above k

From maps to sets



$\overline{F}_k = \{x \in E, F(x) < k\}$, *i.e.*, points below k

Structuring a discrete map: level sets decomposition

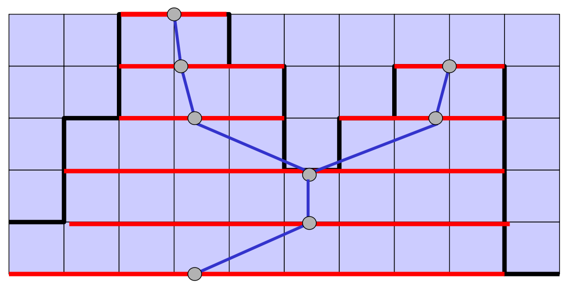
Definition

- *Let F be a map from V to N and let k be an integer*
- *The* **(upper) level-set of F at level k** *is the set made of the points of V whose value is greater than or equal to k*
  - $F_k = \{x \in E, F(x) \geq k\}$
- *Any connected component of any (upper) level set of F is called a* **(upper) component of F**

Remark

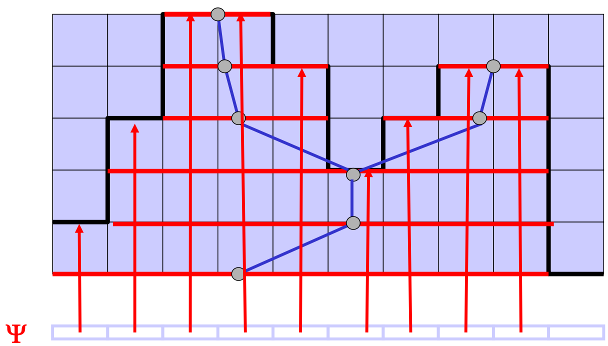
- Any two components of F are either nested or disjoint

Structuring a discrete map: *(upper)* *Component tree*



Components + inclusion relation → tree structure

Structuring a discrete map: ***(upper)*** *Component tree*



Level-sets, components: the 2D case

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

F

Level-sets, components: the 2D case

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

F

1	1	1	1	1	1	1
1	1	1	1	1	1	1
1	1	1	1	1	1	1
1	1	1	1	1	1	1
1	1	1	1	1	1	1
1	1	1	1	1	1	1
1	1	1	1	1	1	1

F_1

Level-sets, components: the 2D case

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

F

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

F_2

Level-sets, components: the 2D case

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

F

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

F_3

Level-sets, components: the 2D case

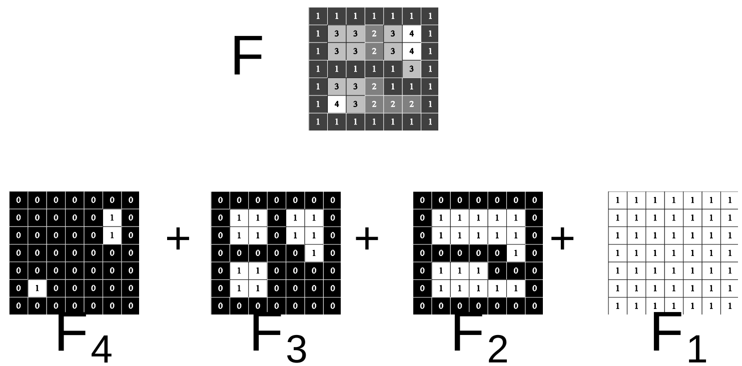
1	1	1	1	1	1	1
1	3	3	2	3	4	1
1	3	3	2	3	4	1
1	1	1	1	1	3	1
1	3	3	2	1	1	1
1	4	3	2	2	2	1
1	1	1	1	1	1	1

F

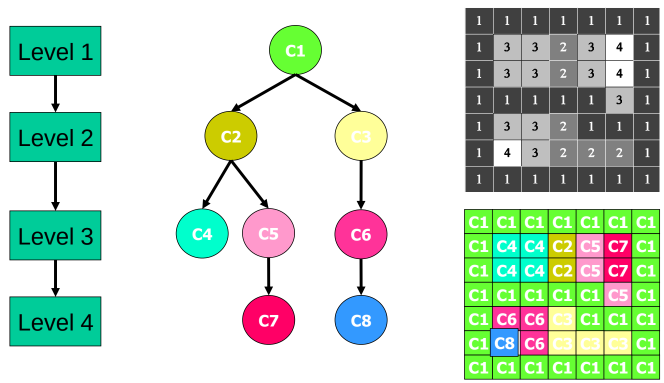
0	0	0	0	0	0	0
0	0	0	0	0	1	0
0	0	0	0	0	1	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	1	0	0	0	0	0
0	0	0	0	0	0	0

F_4

Cross-sections, components: the 2D case



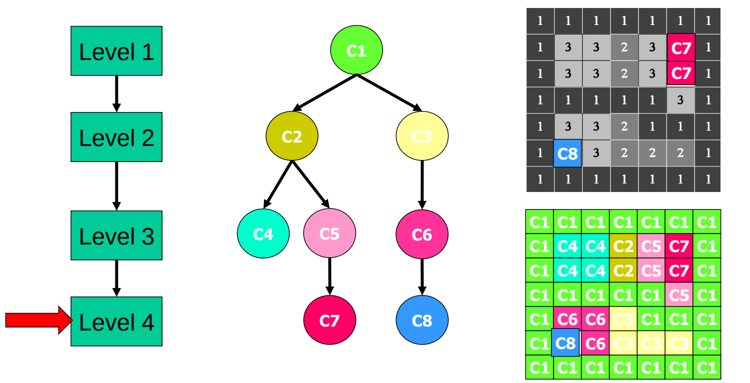
Component tree: the 2D case



Component tree

Component mapping

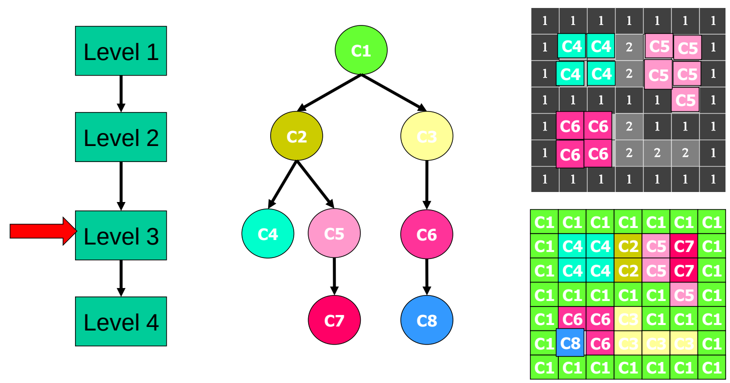
Component tree: the 2D case



Component tree

Component mapping

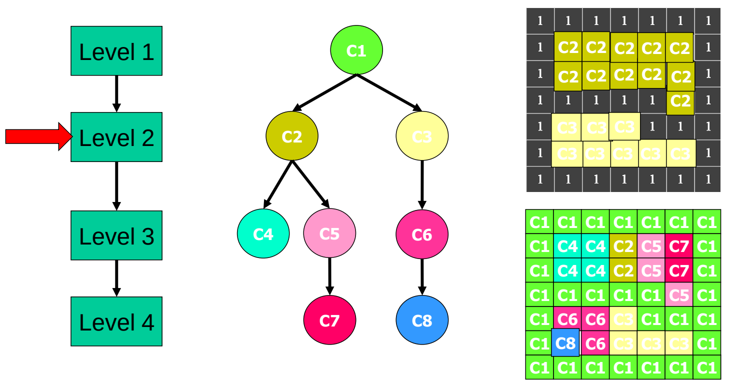
Component tree: the 2D case



Component tree

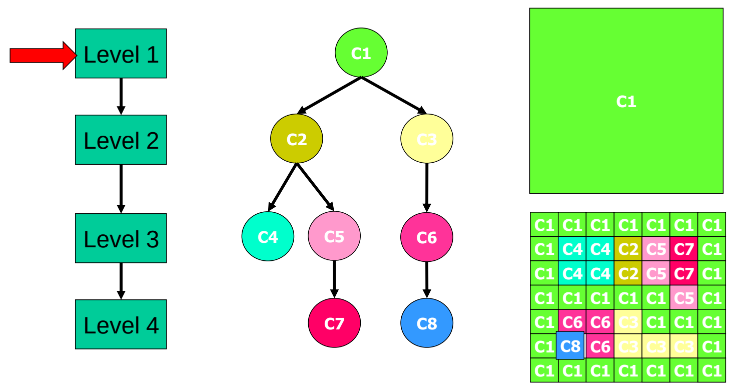
Component mapping

Component tree: the 2D case



Component tree   Component mapping

Component tree: the 2D case

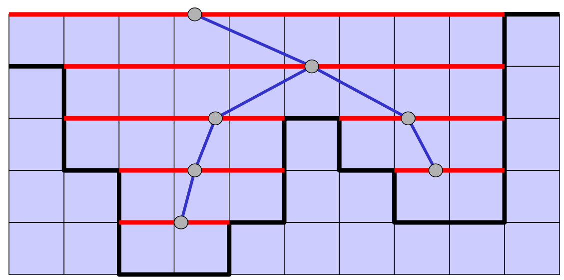


Component tree   Component mapping

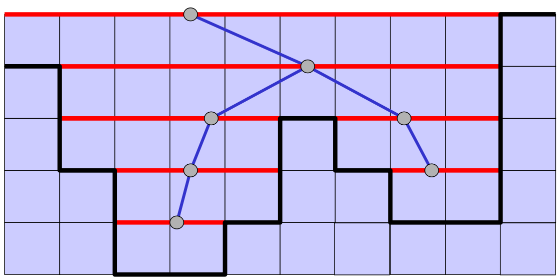
Filtering a discrete map : flooding

- A leaf of the component tree T of F is:
  - a ***regional minimum of F,*** if T is a lower component tree
  - a ***regional maximum of F,*** if T is an upper component tree
- According to a given criterion, we can iteratively remove the leaf components of F
- Such a removal is called a flooding

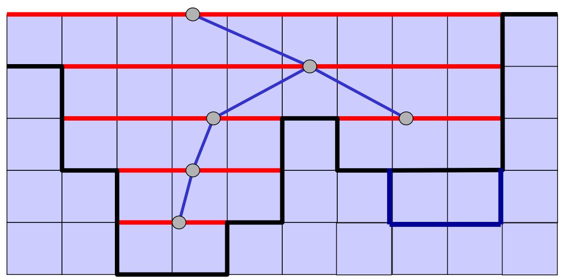
Flooding: intuitive example step 1



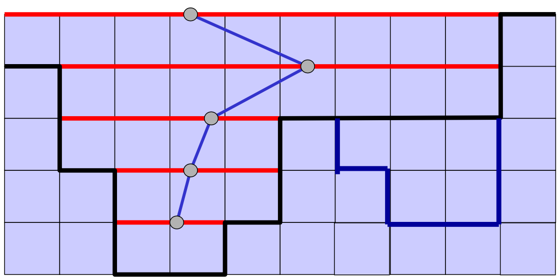
Flooding: intuitive example step 1



Flooding: intuitive example step 2



Flooding: intuitive example step 3



We remove one branch of T
We flood one “ lobe ” of F

Flooding: formal definition

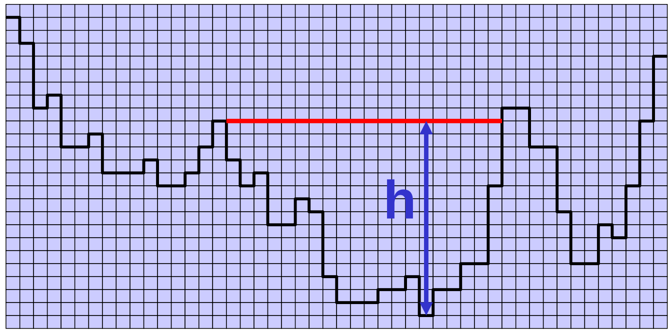
Definition

- *Let x in V*
- *The* **elementary flooding of *F* at *x*** *is defined by:*
  - $\eta_x F(y) = F(y) + 1$ *if x and y belong to the same minimum*
  - $\eta_x F(y) = F(y)$ *otherwise*
- *Any map obtained from F by composition of elementary floodings is called a* **flooding of F**

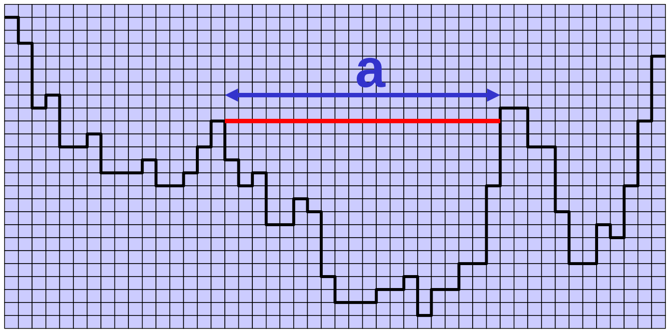
Filtering a grayscale map

- What criterion (attribute) to select the non-significant minima to be flooded?

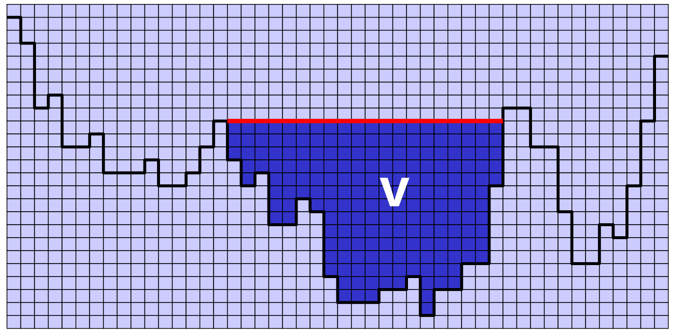
Attributes: height



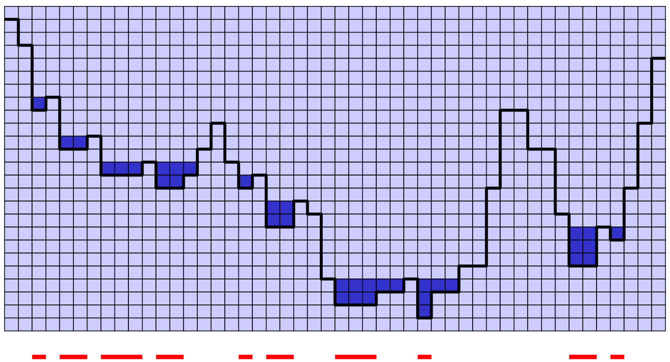
Attributes: area



Attributes: volume

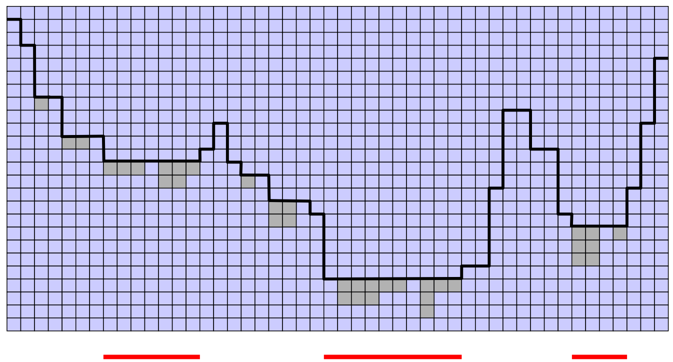


Filtering by volume



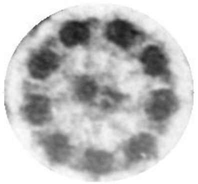
Regional minima

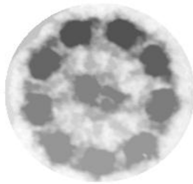
Filtering by volume



Regional minima

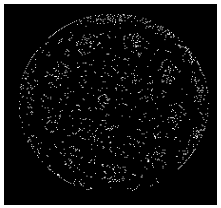
Application: keep the 10 most volumic lobes

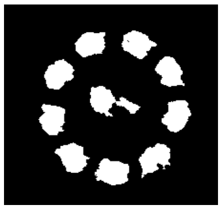




Original image (top) and its many minima (bottom)

Filtered image (top) and its 10 minima (bottom)





Algorithms requested for achieving this result

- **Component tree algorithm**
- Flooding the minima with attribute below a given threshold
- Or alternatively, keeping only the *k* most significant minima

Algorithms for the component tree

- There exist several algorithms for the component tree
  - Breen & Jones 1996
  - Salembier, Oliveras & Garido 1998
  - Mattes & Demongeot 2000
  - **Couprie & Najman 2006**

- **Tarjan's union find**
- **Connected components labeling**
- **Quasi-linear time component tree extraction**

Disjoint set problem

- Maintaining a collection of **disjoint subsets** of a set *V* under the **operation of union**

- Each subset *X* is represented by a unique element of X called the *canonical element of X*

- Three operations allow the collection to be managed:
  - **MakeSet(x)**: add the set {x} to the collection
  - **Find(x):** return the canonical element of the set containing x
  - **Link(x,y):**
    - Remove the two sets whose canonical elements are *x* and *y*
    - Add their union

Tarjan's union-find algorithm

```
Procedure MakeSet(element x)
    Fth(x) := x;  Rnk(x) := 0;
```

```
Function element Find (element x)
    if (Fth(x) ≠ x) then Fth(x) := Find(Fth(x));
    return Fth(x);
```

```
Function element Link (element x, element y)
    if (Rnk(x) > Rnk(y)) then exchange(x, y);
    if (Rnk(x) == Rnk(y)) then Rnk(y) := Rnk(y) + 1;
    Fth(x) := y;
    return y;
```

- **Quasi-linear algorithm**: ***O(m X α(m,|V|))***
  - for m operations
  - ***α*** : inverse of Ackerman function (in practice never greater than 4)

Connected components labeling with union-find

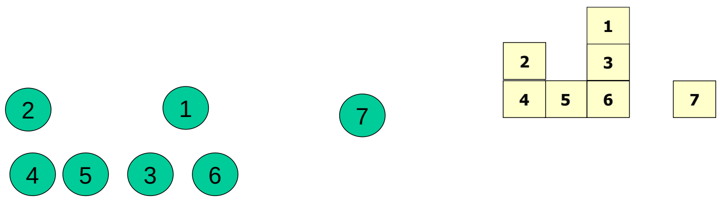
```
Algorithm 1: BuildConnectedComponents
  Data: (E, Γ) - graph
  Data: A set X ⊆ E
  Result: M - map from X to E
foreach p ∈ X do MakeSet(p);
foreach p ∈ X do
    compp := Find(p);
    foreach q ∈ Γ(p) such that q ∈ X do
        compq := Find(q);
        if (compp ≠ compq) then compp := Link(compq, compp);
for p ∈ X do M(p) := Find(p);
```

Connected components labeling with union-find

Algorithm 1: BuildConnectedComponents

Data: (E, Γ) - graph
Data: A set $X \subseteq E$
Result: M - map from X to E

foreach $p \in X$ **do** MakeSet(p);
foreach $p \in X$ **do**
  compp := Find(p);
  foreach $q \in \Gamma(p)$ *such that* $q \in X$ **do**
    compq := Find(q);
    if *(compp* $\neq$ *compq)* **then** compp := Link(compq, compp);
for $p \in X$ **do** M(p) := Find(p);

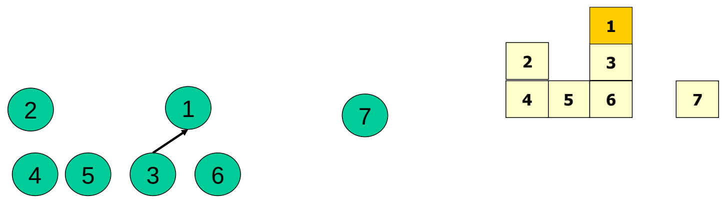


Connected components labeling with union-find

Algorithm 1: BuildConnectedComponents

Data: (E, Γ) - graph
Data: A set $X \subseteq E$
Result: M - map from X to E

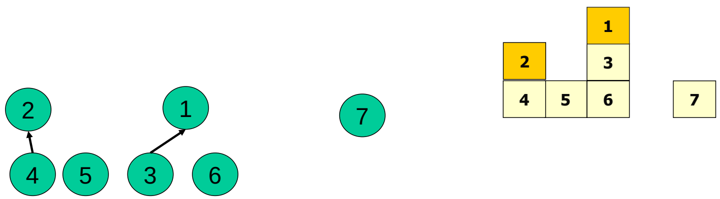
foreach $p \in X$ **do** MakeSet(p);
foreach $p \in X$ **do**
  compp := Find(p);
  foreach $q \in \Gamma(p)$ *such that* $q \in X$ **do**
    compq := Find(q);
    if *(compp* $\neq$ *compq)* **then** compp := Link(compq, compp);
for $p \in X$ **do** M(p) := Find(p);



Connected components labeling with union-find

Algorithm 1: BuildConnectedComponents

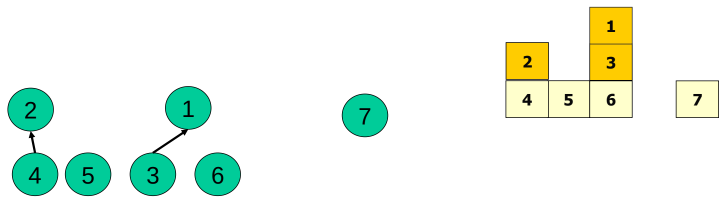
```
Data: (E, Γ) - graph
Data: A set X ⊆ E
Result: M - map from X to E
foreach p ∈ X do MakeSet(p);
foreach p ∈ X do
    compp := Find(p);
    foreach q ∈ Γ(p) such that q ∈ X do
        compq := Find(q);
        if (compp ≠ compq) then compp := Link(compq, compp);
for p ∈ X do M(p) := Find(p);
```



Connected components labeling with union-find

Algorithm 1: BuildConnectedComponents

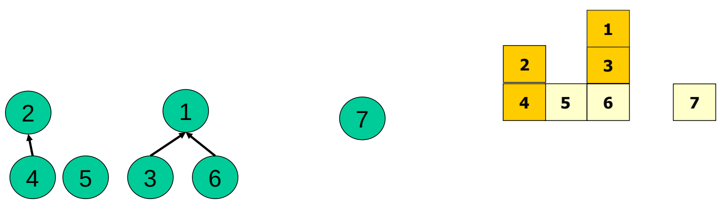
```
Data: (E, Γ) - graph
Data: A set X ⊆ E
Result: M - map from X to E
foreach p ∈ X do MakeSet(p);
foreach p ∈ X do
    compp := Find(p);
    foreach q ∈ Γ(p) such that q ∈ X do
        compq := Find(q);
        if (compp ≠ compq) then compp := Link(compq, compp);
for p ∈ X do M(p) := Find(p);
```



Connected components labeling with union-find

Algorithm 1: BuildConnectedComponents

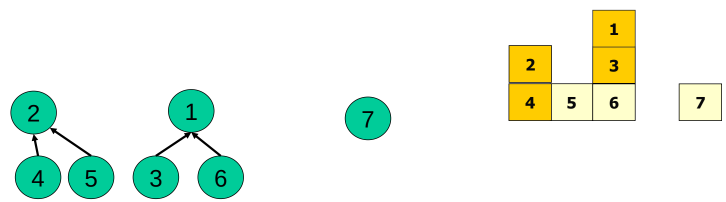
```
    Data: (E, Γ) - graph
    Data: A set X ⊆ E
    Result: M - map from X to E
foreach p ∈ X do MakeSet(p);
foreach p ∈ X do
    compp := Find(p);
    foreach q ∈ Γ(p) such that q ∈ X do
        compq := Find(q);
        if (compp ≠ compq) then compp := Link(compq, compp);
for p ∈ X do M(p) := Find(p);
```



Connected components labeling with union-find

Algorithm 1: BuildConnectedComponents

```
    Data: (E, Γ) - graph
    Data: A set X ⊆ E
    Result: M - map from X to E
foreach p ∈ X do MakeSet(p);
foreach p ∈ X do
    compp := Find(p);
    foreach q ∈ Γ(p) such that q ∈ X do
        compq := Find(q);
        if (compp ≠ compq) then compp := Link(compq, compp);
for p ∈ X do M(p) := Find(p);
```



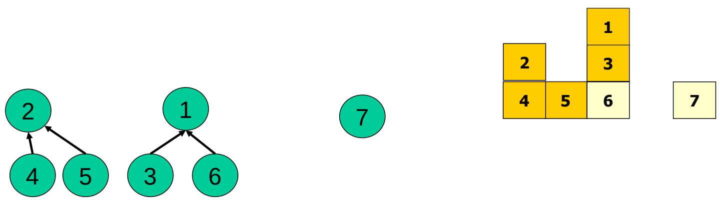
Connected components labeling with union-find

Algorithm 1: BuildConnectedComponents

Data: (E, Γ) - graph
Data: A set $X \subseteq E$
Result: M - map from X to E

foreach $p \in X$ **do** MakeSet(p);
foreach $p \in X$ **do**
  compp := Find(p);
  foreach $q \in \Gamma(p)$ *such that* $q \in X$ **do**
    compq := Find(q);
    if *(compp $\neq$ compq)* **then** compp := Link(compq, compp);

for $p \in X$ **do** M(p) := Find(p);



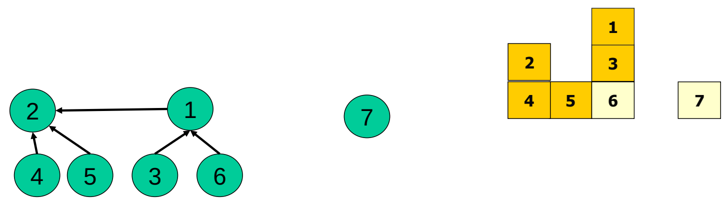
Connected components labeling with union-find

Algorithm 1: BuildConnectedComponents

Data: (E, Γ) - graph
Data: A set $X \subseteq E$
Result: M - map from X to E

foreach $p \in X$ **do** MakeSet(p);
foreach $p \in X$ **do**
  compp := Find(p);
  foreach $q \in \Gamma(p)$ *such that* $q \in X$ **do**
    compq := Find(q);
    if *(compp $\neq$ compq)* **then** compp := Link(compq, compp);

for $p \in X$ **do** M(p) := Find(p);



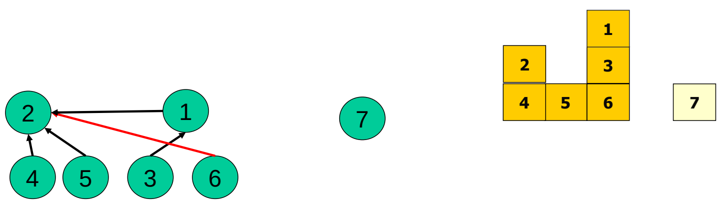
Connected components labeling with union-find

Algorithm 1: BuildConnectedComponents

Data: (E, Γ) - graph
Data: A set $X \subseteq E$
Result: M - map from X to E

foreach $p \in X$ **do** MakeSet(p);
foreach $p \in X$ **do**
  compp := Find(p);
  foreach $q \in \Gamma(p)$ *such that* $q \in X$ **do**
    compq := Find(q);
    if *(compp $\neq$ compq)* **then** compp := Link(compq, compp);

for $p \in X$ **do** M(p) := Find(p);



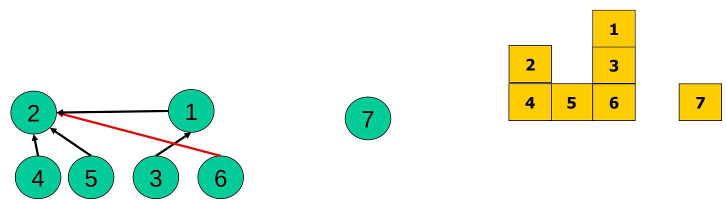
Connected components labeling with union-find

Algorithm 1: BuildConnectedComponents

Data: (E, Γ) - graph
Data: A set $X \subseteq E$
Result: M - map from X to E

foreach $p \in X$ **do** MakeSet(p);
foreach $p \in X$ **do**
  compp := Find(p);
  foreach $q \in \Gamma(p)$ *such that* $q \in X$ **do**
    compq := Find(q);
    if *(compp $\neq$ compq)* **then** compp := Link(compq, compp);

for $p \in X$ **do** M(p) := Find(p);

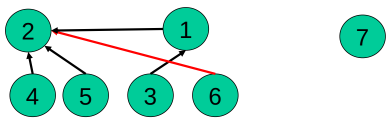


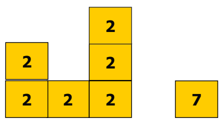
Connected components labeling with union-find

Algorithm 1: BuildConnectedComponents

Data: (E, Γ) - graph
Data: A set $X \subseteq E$
Result: M - map from X to E

foreach $p \in X$ **do** MakeSet(p);
foreach $p \in X$ **do**
  compp := Find(p);
  foreach $q \in \Gamma(p)$ *such that* $q \in X$ **do**
    compq := Find(q);
    if *(compp* $\neq$ *compq)* **then** compp := Link(compq, compp);
for $p \in X$ **do** M(p) := Find(p);





Component tree algorithm

- General case
  - Two collections of disjoint subsets of V
- Particular case: all values are distinct
  - One collection of disjoint subsets of V

Component tree algorithm: all values distinct

- **foreach** point x in V in decreasing order of altitude **do**
  - **foreach** neighbor y of x **do**
    - If (F(y) > F(x)) and (Find(x) != Find(y)) do
      - adjNode := lowestNode[Find(y)]
      - nodes[x].addChild(nodes[adjNode])
      - curNode := Link(x,adjNode)
      - lowestNode[curNode] := nodes[x]

Component tree algorithm: all values distinct (init)

- **foreach** point x in V in decreasing order of altitude **do**
  - **foreach** neighbor y of x **do**
    - If (F(y) > F(x)) and (Find(x) != Find(y)) do
      - adjNode := lowestNode[Find(y)]
      - nodes[x].addChild(nodes[adjNode])
      - curNode := Link(x,adjNode)
      - lowestNode[curNode] := nodes[x]

Image

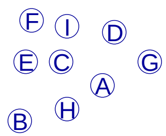
A:7	B:1	C:9
D:6	E:4	F:2
G:8	H:5	I:3

Lowest Node

A	B	C
D	E	F
G	H	I

nodes

Disjoint sets



Component tree algorithm: all values distinct (step 1)

- **foreach** point x in V in decreasing order of altitude **do**
  - **foreach** neighbor y of x **do**
    - If (F(y) > F(x)) and (Find(x) != Find(y)) do
      - adjNode := lowestNode[Find(y)]
      - nodes[x].addChild(nodes[adjNode])
      - curNode := Link(x,adjNode)
      - lowestNode[curNode] := nodes[x]

x = C

Image

A:7	B:1	C:9
D:6	E:4	F:2
G:8	H:5	I:3

Lowest Node

A	B	C
D	E	F
G	H	I

nodes

C

Disjoint sets

F I D
E C G
A
B H

Component tree algorithm: all values distinct (step 2)

- **foreach** point x in V in decreasing order of altitude **do**
  - **foreach** neighbor y of x **do**
    - If (F(y) > F(x)) and (Find(x) != Find(y)) do
      - adjNode := lowestNode[Find(y)]
      - nodes[x].addChild(nodes[adjNode])
      - curNode := Link(x,adjNode)
      - lowestNode[curNode] := nodes[x]

x = G

Image

A:7	B:1	C:9
D:6	E:4	F:2
G:8	H:5	I:3

Lowest Node

A	B	C
D	E	F
G	H	I

nodes

G C

Disjoint sets

F I D
E C G
A
B H

Component tree algorithm: all values distinct (step 3)

- **foreach** point x in V in decreasing order of altitude **do**
  - **foreach** neighbor y of x **do**
    - If (F(y) > F(x)) and (Find(x) != Find(y)) do
      - adjNode := lowestNode[Find(y)]
      - nodes[x].addChild(nodes[adjNode])
      - curNode := Link(x,adjNode)
      - lowestNode[curNode] := nodes[x]

x = A

Image

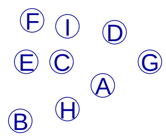
A:7	B:1	C:9
D:6	E:4	F:2
G:8	H:5	I:3

Lowest Node

A	B	C
D	E	F
G	H	I

nodes

Disjoint sets



Component tree algorithm: all values distinct (step 4)

- **foreach** point x in V in decreasing order of altitude **do**
  - **foreach** neighbor y of x **do**
    - If (F(y) > F(x)) and (Find(x) != Find(y)) do
      - adjNode := lowestNode[Find(y)]
      - nodes[x].addChild(nodes[adjNode])
      - curNode := Link(x,adjNode)
      - lowestNode[curNode] := nodes[x]

x = D

Image

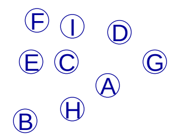
A:7	B:1	C:9
D:6	E:4	F:2
G:8	H:5	I:3

Lowest Node

A	B	C
D	E	F
G	H	I

nodes

Disjoint sets



Component tree algorithm: all values distinct (step 4)

- **foreach** point x in V in decreasing order of altitude **do**
  - **foreach** neighbor y of x **do**
    - If (F(y) > F(x)) and (Find(x) != Find(y)) do
      - adjNode := lowestNode[Find(y)]
      - nodes[x].addChild(nodes[adjNode])
      - curNode := Link(x,adjNode)
      - lowestNode[curNode] := nodes[x]

x = D
y = A

Image

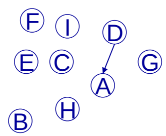
A:7	B:1	C:9
D:6	E:4	F:2
G:8	H:5	I:3

Lowest Node

D	B	C
D	E	F
G	H	I

nodes

Disjoint sets



Component tree algorithm: all values distinct (step 4)

- **foreach** point x in V in decreasing order of altitude **do**
  - **foreach** neighbor y of x **do**
    - If (F(y) > F(x)) and (Find(x) != Find(y)) do
      - adjNode := lowestNode[Find(y)]
      - nodes[x].addChild(nodes[adjNode])
      - curNode := Link(x,adjNode)
      - lowestNode[curNode] := nodes[x]

x = D
y = G

Image

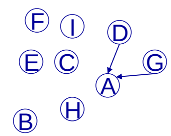
A:7	B:1	C:9
D:6	E:4	F:2
G:8	H:5	I:3

Lowest Node

D	B	C
D	E	F
G	H	I

nodes

Disjoint sets



Component tree algorithm: all values distinct (step 5)

- **foreach** point x in V in decreasing order of altitude **do**
  - **foreach** neighbor y of x **do**
    - If (F(y) > F(x)) and (Find(x) != Find(y)) do
      - adjNode := lowestNode[Find(y)]
      - nodes[x].addChild(nodes[adjNode])
      - curNode := Link(x,adjNode)
      - lowestNode[curNode] := nodes[x]

x = H
y = G

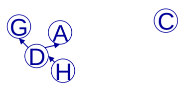
Image

A:7	B:1	C:9
D:6	E:4	F:2
G:8	H:5	I:3

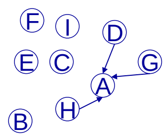
Lowest Node

H	B	C
D	E	F
G	H	I

nodes



Disjoint sets



Component tree algorithm: all values distinct (step 6)

- **foreach** point x in V in decreasing order of altitude **do**
  - **foreach** neighbor y of x **do**
    - If (F(y) > F(x)) and (Find(x) != Find(y)) do
      - adjNode := lowestNode[Find(y)]
      - nodes[x].addChild(nodes[adjNode])
      - curNode := Link(x,adjNode)
      - lowestNode[curNode] := nodes[x]

x = E
y = D

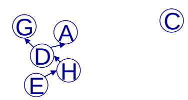
Image

A:7	B:1	C:9
D:6	E:4	F:2
G:8	H:5	I:3

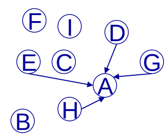
Lowest Node

E	B	C
D	E	F
G	H	I

nodes



Disjoint sets



Component tree algorithm: all values distinct (step 6)

- **foreach** point x in V in decreasing order of altitude **do**
  - **foreach** neighbor y of x **do**
    - If (F(y) > F(x)) and (Find(x) != Find(y)) do
      - adjNode := lowestNode[Find(y)]
      - nodes[x].addChild(nodes[adjNode])
      - curNode := Link(x,adjNode)
      - lowestNode[curNode] := nodes[x]

x = E
y = H

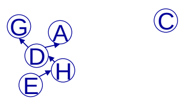
Image

A:7	B:1	C:9
D:6	E:4	F:2
G:8	H:5	I:3

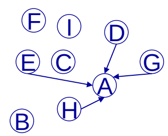
Lowest Node

E	B	C
D	E	F
G	H	I

nodes



Disjoint sets



Component tree algorithm: all values distinct (step 7)

- **foreach** point x in V in decreasing order of altitude **do**
  - **foreach** neighbor y of x **do**
    - If (F(y) > F(x)) and (Find(x) != Find(y)) do
      - adjNode := lowestNode[Find(y)]
      - nodes[x].addChild(nodes[adjNode])
      - curNode := Link(x,adjNode)
      - lowestNode[curNode] := nodes[x]

x = I
y = H

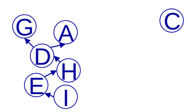
Image

A:7	B:1	C:9
D:6	E:4	F:2
G:8	H:5	I:3

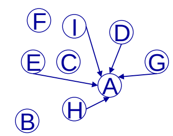
Lowest Node

I	B	C
D	E	F
G	H	I

nodes



Disjoint sets



Component tree algorithm: all values distinct (step 8)

- **foreach** point x in V in decreasing order of altitude **do**
  - **foreach** neighbor y of x **do**
    - If (F(y) > F(x)) and (Find(x) != Find(y)) do
      - adjNode := lowestNode[Find(y)]
      - nodes[x].addChild(nodes[adjNode])
      - curNode := Link(x,adjNode)
      - lowestNode[curNode] := nodes[x]

x = F
y = C

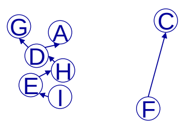
Image

A:7	B:1	C:9
D:6	E:4	F:2
G:8	H:5	I:3

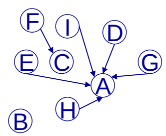
Lowest Node

I	B	F
D	E	F
G	H	I

nodes



Disjoint sets



Component tree algorithm: all values distinct (step 7)

- **foreach** point x in V in decreasing order of altitude **do**
  - **foreach** neighbor y of x **do**
    - If (F(y) > F(x)) and (Find(x) != Find(y)) do
      - adjNode := lowestNode[Find(y)]
      - nodes[x].addChild(nodes[adjNode])
      - curNode := Link(x,adjNode)
      - lowestNode[curNode] := nodes[x]

x = F
y = E

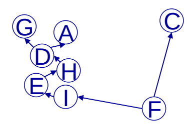
Image

A:7	B:1	C:9
D:6	E:4	F:2
G:8	H:5	I:3

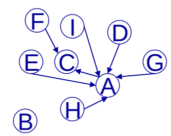
Lowest Node

I	B	F
D	E	F
G	H	I

nodes



Disjoint sets



Component tree algorithm: all values distinct (step 7)

- **foreach** point x in V in decreasing order of altitude **do**
  - **foreach** neighbor y of x **do**
    - If (F(y) > F(x)) and (Find(x) != Find(y)) do
      - adjNode := lowestNode[Find(y)]
      - nodes[x].addChild(nodes[adjNode])
      - curNode := Link(x,adjNode)
      - lowestNode[curNode] := nodes[x]

x = F
y = I

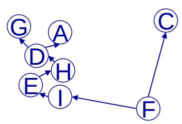
Image

A:7	B:1	C:9
D:6	E:4	F:2
G:8	H:5	I:3

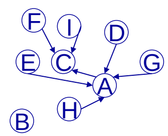
Lowest Node

I	B	F
D	E	F
G	H	I

nodes



Disjoint sets



Component tree algorithm: all values distinct (step 9)

- **foreach** point x in V in decreasing order of altitude **do**
  - **foreach** neighbor y of x **do**
    - If (F(y) > F(x)) and (Find(x) != Find(y)) do
      - adjNode := lowestNode[Find(y)]
      - nodes[x].addChild(nodes[adjNode])
      - curNode := Link(x,adjNode)
      - lowestNode[curNode] := nodes[x]

x = B
y = A

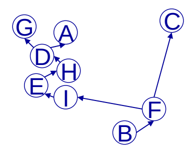
Image

A:7	B:1	C:9
D:6	E:4	F:2
G:8	H:5	I:3

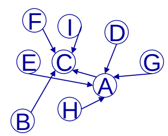
Lowest Node

I	B	B
D	E	F
G	H	I

nodes



Disjoint sets



Component tree algorithm: all values distinct (step 10)

- **foreach** point x in V in decreasing order of altitude **do**
  - **foreach** neighbor y of x **do**
    - If (F(y) > F(x)) and (Find(x) != Find(y)) do
      - adjNode := lowestNode[Find(y)]
      - nodes[x].addChild(nodes[adjNode])
      - curNode := Link(x,adjNode)
      - lowestNode[curNode] := nodes[x]

x = B
y = C

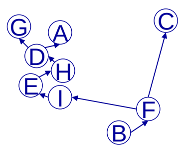
Image

A:7	B:1	C:9
D:6	E:4	F:2
G:8	H:5	I:3

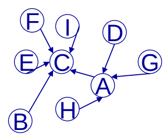
Lowest Node

I	B	B
D	E	F
G	H	I

nodes



Disjoint sets



Component tree algorithm: all values distinct

- **foreach** point x in V in decreasing order of altitude **do** *O(|V|)*
  - **foreach** neighbor y of x **do** *O(|V|+|E|)*
    - If (F(y) > F(x)) and (Find(x) != Find(y)) do *O(|E|.α(|E|,|V|))*
      - adjNode := lowestNode[Find(y)] *O(|E|)*
      - nodes[x].addChild(nodes[adjNode]) *O(|E|)*
      - curNode := Link(x,adjNode) *O(|E|. α(|E|,|V|))*
      - lowestNode[curNode] := nodes[x] *O(|E|)*

- **Time complexity : quasi-linear**
  - **with respect to the number of edges and vertices**
  - **Requires a sorting pre-processing step**

Component tree algorithm: general case

- Two union-find implementations needed
  - One for building the union of connected components over height
    - Similar to the one in the case where all values were distinct
  - One for building the connected components of a given altitude

Build the component tree

Algorithm 2: BuildComponentTree

Data: (E, Γ, F) - weighted graph with N points.
Result: N_n - number of nodes of the component tree ($\leq N$).
Result: $nodes$ - array $[0 \ldots N-1]$ of nodes.
Result: $Root$ - Root of the component tree
Result: C - map from E to $[0 \ldots N-1]$ (component mapping).
Local: $subtreeRoot$ - map from $[0 \ldots N-1]$ to $[0 \ldots N-1]$.

```
N_n := N; Sort the points in decreasing order of level for F;
foreach p ∈ E do {MakeSet1(p); MakeSet2(p); nodes[p]:= MakeNode(F(p)); subtreeRoot[p] := p;};
foreach p ∈ E in decreasing order of level for F do
    curCanonicalElt := Find1(p);
    curNode := Find2(subtreeRoot[curCanonicalElt]);
    foreach already processed neighbor q of p with F(q) ≥ F(p) do
        adjCanonicalElt := Find1(q);
        adjNode := Find2(subtreeRoot[adjCanonicalElt]);
        if (curNode ≠ adjNode) then
            if (nodes[curNode]→level == nodes[adjNode]→level) then
                curNode := MergeNodes(adjNode, curNode);
            else
                //We have nodes[curNode]→level < nodes[adjNode]→level
                nodes[curNode]→addSon(nodes[adjNode]);
            curCanonicalElt := Link1(adjCanonicalElt, curCanonicalElt);
            subtreeRoot[curCanonicalElt] := curNode;
Root := subtreeRoot[Find1(Find2(0))] ;
foreach p ∈ E do C(p) := Find2(p);
```

Auxiliary functions

Function node MakeNode(*int level*)

  allocate a new node n with an empty list of sons;
  n→ level := level;
  return n;

Function int MergeNodes(*int node*1, *int node*2)

  tmpNode := Link2(node1,node2);
  if *(tmpNode == node2)* **then**
    Add the list of sons of *nodes*[node1] to the list of sons of *nodes*[node2];
  else
    Add the list of sons of *nodes*[node2] to the list of sons of *nodes*[node1];
  $N_n := N_n - 1$;
  return tmpNode;

Example

110	90	100
50	50	50
40	20	50
50	50	50
120	70	80

0	1	2
3	4	5
6	7	8
9	10	11
12	13	14

12;	[120]
0;	[110]
2;	[100]
1;	[90]
14;	[80]
13;	[70]
3; 4; 5; 8; 9; 10; 11;	[50]
6;	[40]
7	[20]

step 1

110	90	100
50	50	50
40	20	50
50	50	50
120	70	80

Fth1

0	1	2
3	4	5
6	7	8
9	10	11
12	13	14

Fth2

0	1	2
3	4	5
6	7	8
9	10	11
12	13	14

SubtreeRoot

0	1	2
3	4	5
6	7	8
9	10	11
12	13	14

[12] 120

Step 2

110	90	100
50	50	50
40	20	50
50	50	50
120	70	80

Fth1

0	1	1
3	4	5
6	7	8
9	10	11
12	13	14

Fth2

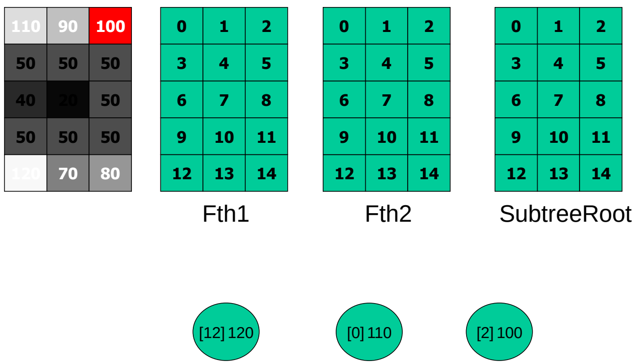
0	1	2
3	4	5
6	7	8
9	10	11
12	13	14

SubtreeRoot

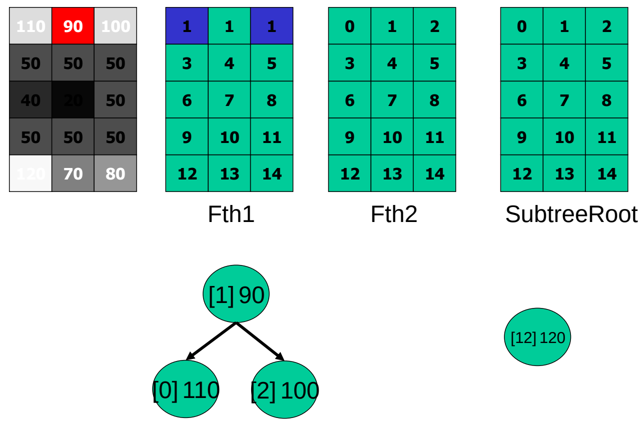
0	1	2
3	4	5
6	7	8
9	10	11
12	13	14

[12] 120 [0] 110

step 3


110 90 100
50 50 50
40 20 50
50 50 50
120 70 80
Fth1
Fth2
SubtreeRoot
[12] 120
[0] 110
[2] 100

step 4


110 90 100
50 50 50
40 20 50
50 50 50
120 70 80
Fth1
Fth2
SubtreeRoot
[1] 90
[0] 110
[2] 100
[12] 120

step 5

110	90	100
50	50	50
40	20	50
50	50	50
120	70	80

Fth1

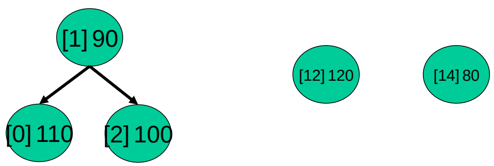
1	1	1
3	4	5
6	7	8
9	10	11
12	13	14

Fth2

0	1	2
3	4	5
6	7	8
9	10	11
12	13	14

SubtreeRoot

0	1	2
3	4	5
6	7	8
9	10	11
12	13	14



step 6

110	90	100
50	50	50
40	20	50
50	50	50
120	70	80

Fth1

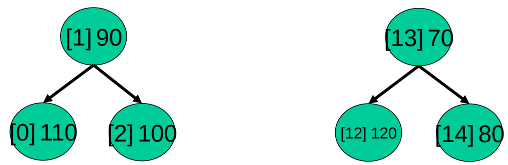
1	1	1
3	4	5
6	7	8
9	10	11
13	13	13

Fth2

0	1	2
3	4	5
6	7	8
9	10	11
12	13	14

SubtreeRoot

0	1	2
3	4	5
6	7	8
9	10	11
12	13	14



step 7

110	90	100
50	50	50
40	20	50
50	50	50
120	70	80

1	3	1
3	4	5
6	7	8
9	10	11
13	13	13

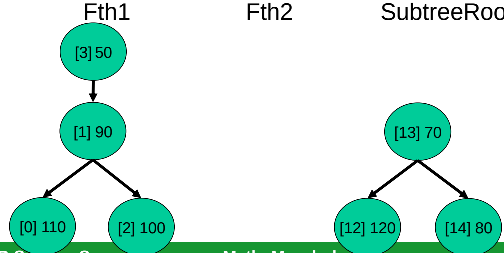
Fth1

0	1	2
3	4	5
6	7	8
9	10	11
12	13	14

Fth2

0	1	2
3	4	5
6	7	8
9	10	11
12	13	14

SubtreeRoot



step 8

110	90	100
50	50	50
40	20	50
50	50	50
120	70	80

1	3	1
3	3	5
6	7	8
9	10	11
13	13	13

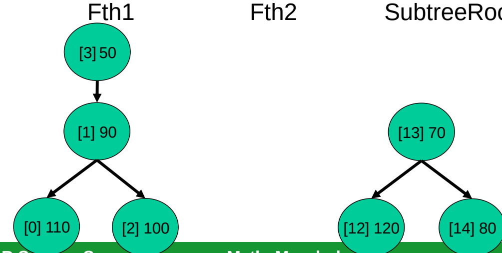
Fth1

0	1	2
3	3	5
6	7	8
9	10	11
12	13	14

Fth2

0	1	2
3	4	5
6	7	8
9	10	11
12	13	14

SubtreeRoot



step 9

110	90	100
50	50	50
40	20	50
50	50	50
120	70	80

1	3	3
3	3	3
6	7	8
9	10	11
13	13	13

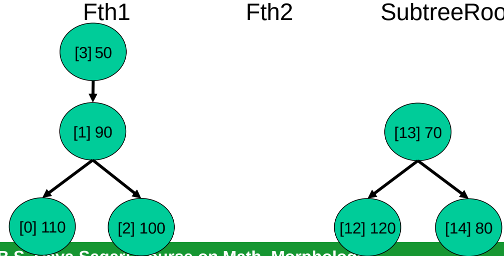
Fth1

0	1	2
3	3	3
6	7	8
9	10	11
12	13	14

Fth2

0	1	2
3	4	5
6	7	8
9	10	11
12	13	14

SubtreeRoot



step 10

110	90	100
50	50	50
40	20	50
50	50	50
120	70	80

1	3	3
3	3	3
6	7	3
9	10	11
13	13	13

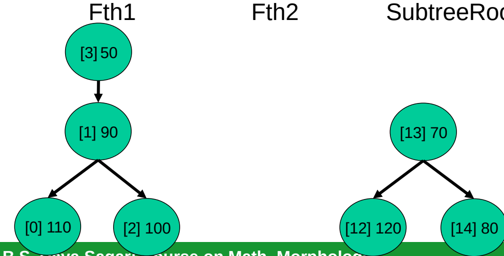
Fth1

0	1	2
3	3	3
6	7	3
9	10	11
12	13	14

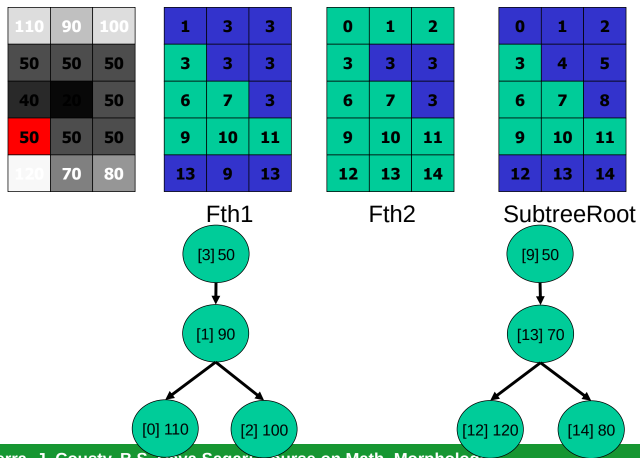
Fth2

0	1	2
3	4	5
6	7	8
9	10	11
12	13	14

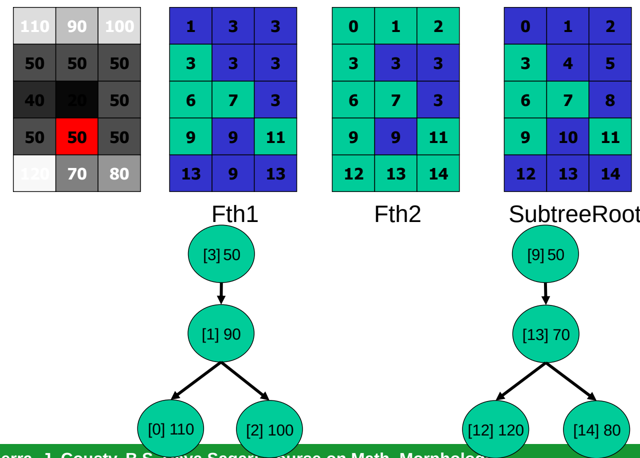
SubtreeRoot



step 11



step 12



step 13

110	90	100
50	50	50
40	20	50
50	50	50
120	70	80

1	3	3
9	3	3
6	7	3
9	9	3
13	9	13

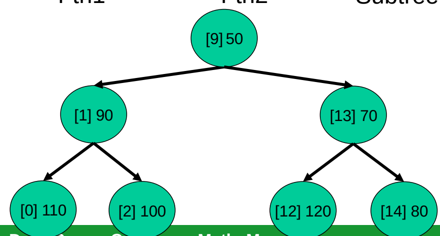
Fth1

0	1	2
9	3	3
6	7	3
9	9	3
12	13	14

Fth2

0	1	2
3	4	5
6	7	8
9	10	11
12	13	14

SubtreeRoot



step 14

110	90	100
50	50	50
40	20	50
50	50	50
120	70	80

1	3	3
9	3	3
9	7	3
9	9	3
13	9	13

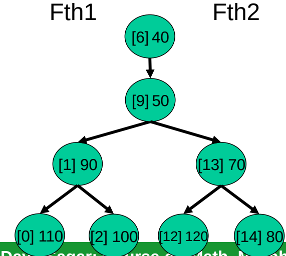
Fth1

0	1	2
9	3	3
6	7	3
9	9	3
12	13	14

Fth2

0	1	2
3	4	5
6	7	8
6	10	11
12	13	14

SubtreeRoot



step 15

110	90	100
50	50	50
40	20	50
50	50	50
120	70	80

1	3	3
9	9	3
9	9	9
9	9	3
13	9	13

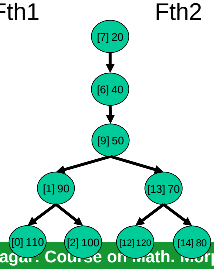
Fth1

0	1	2
9	3	3
6	7	3
9	9	3
12	13	14

Fth2

0	1	2
3	4	5
6	7	8
7	10	11
12	13	14

SubtreeRoot



Component tree: conclusion

- Another insight into connected filters (level sets stacking)
- Useful to compute floodings
  - then attribute openings & closings
- Efficient algorithm
  - Quasi-linear time
  - Easy implementation
- Many applications in signal processing and image analysis
- Important for computing connection values
- There exist self-dual trees