

# Connected Component Tree

**Jean Cousty**

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



**ESIEE**  
ENGINEERING

UNIVERSITÉ  
— PARIS-EST



## 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?**

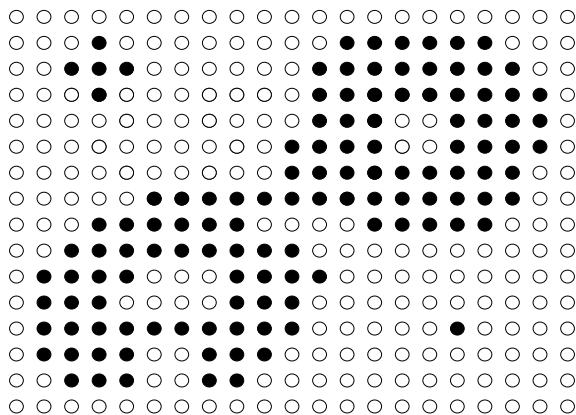
## Outline

- 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

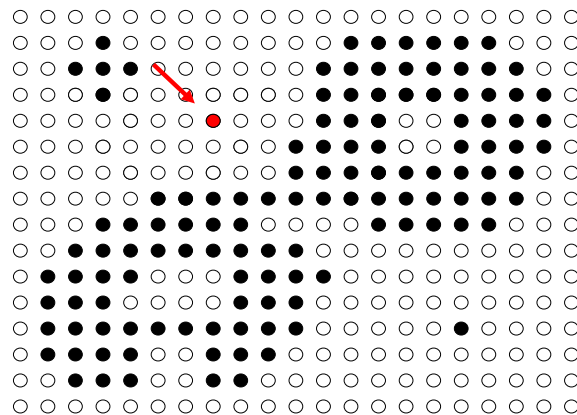
- Let  $V$  be a set
  - (eg, the set of all image pixels)
- Let  $X$  be a subset of  $V$ 
  - (eg, a binary object to be processed)

### Discrete set



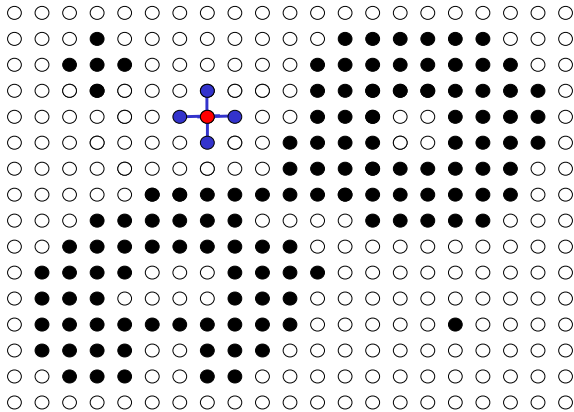
V (white & black dots) and X (black dots)

### Discrete set



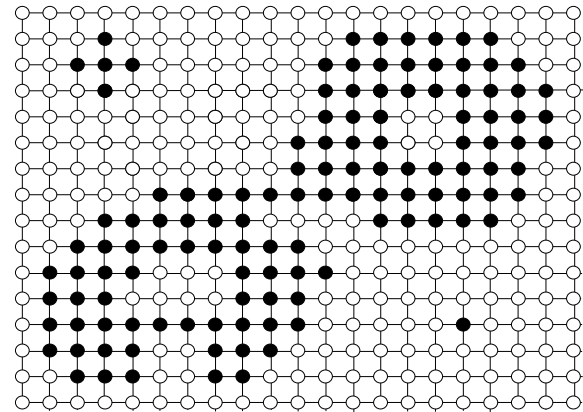
V (white & black dots) and X (black dots)

## Graph



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

## Graph



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

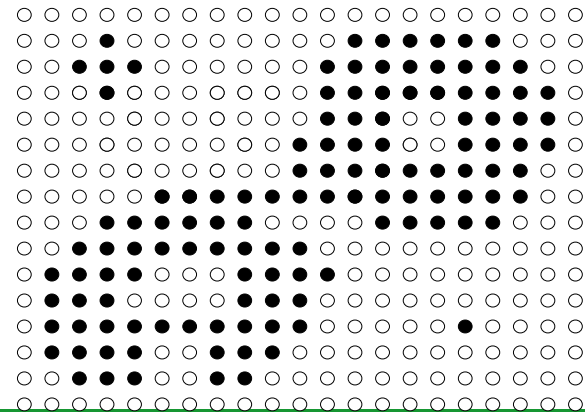
## Connected sets

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

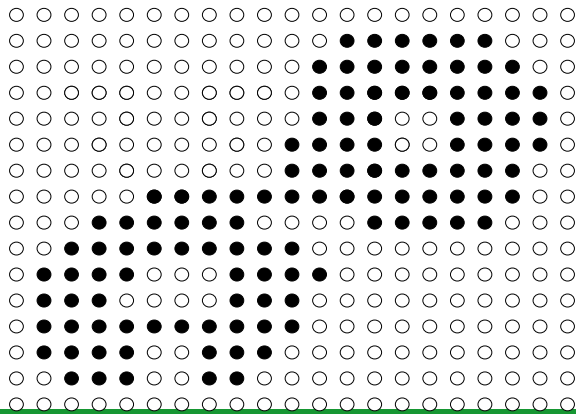
### Definition

- $X$  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  $X$

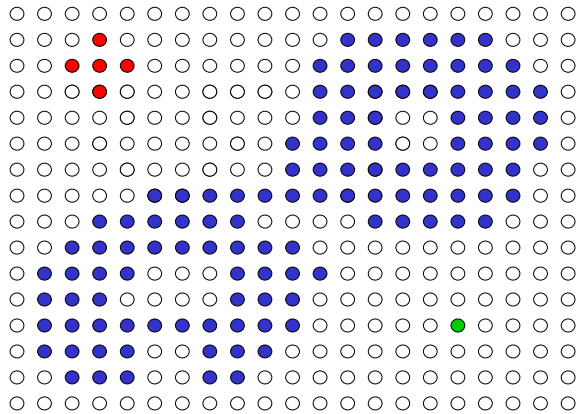
### Definition

- $Y$  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 contains  $x$

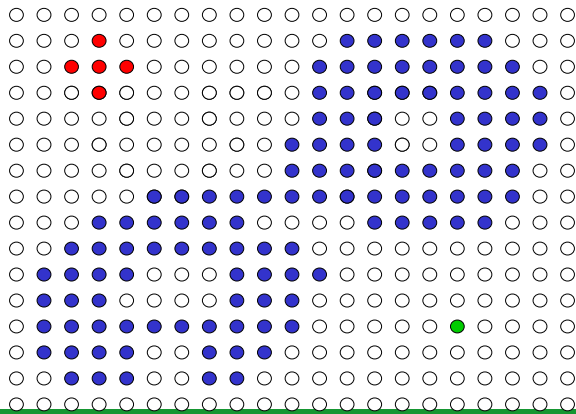
### Example : Connected components partition



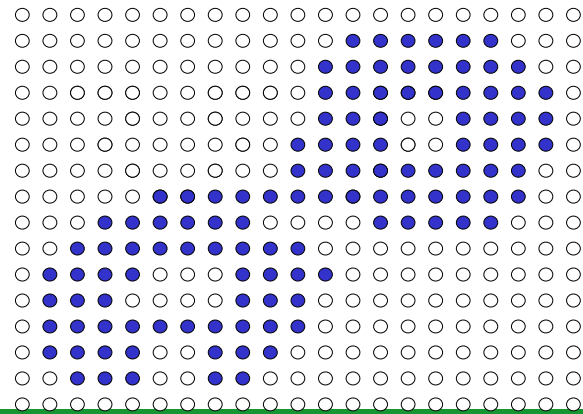
### Connected opening a discrete set

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

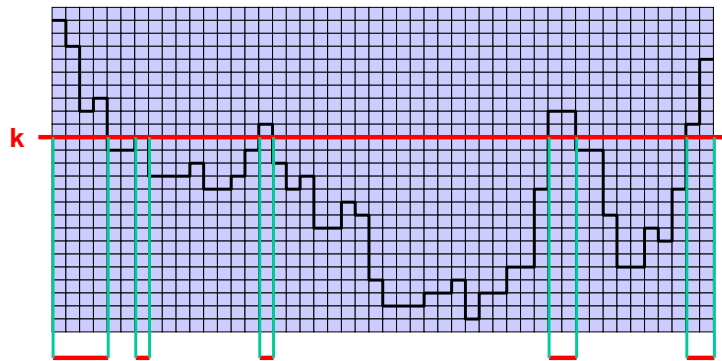
### Example: area filtering a discrete set



### Example: area filtering a discrete set



## From maps to sets

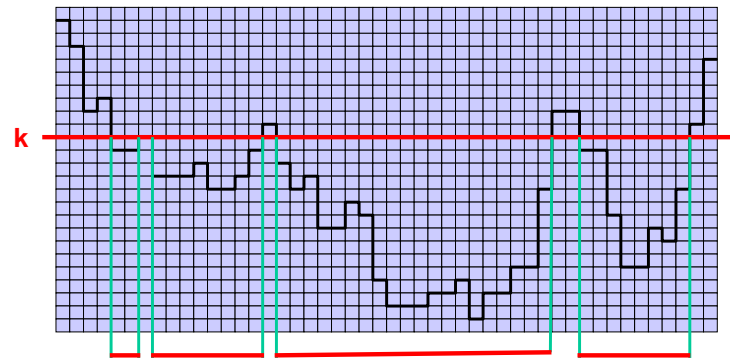


$$F_k = \{x \in E, F(x) \geq k\}, \text{ i.e., points above } k$$

J. Serra, J. Cousty, B.S. Daya Sagar: Course on Math. Morphology

17

## From maps to sets



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

J. Serra, J. Cousty, B.S. Daya Sagar: Course on Math. Morphology

18

## Structuring a discrete map: level sets decomposition

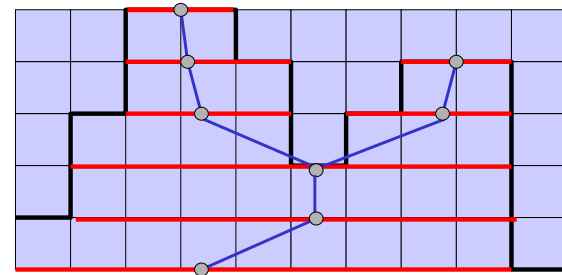
### Definition

- 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

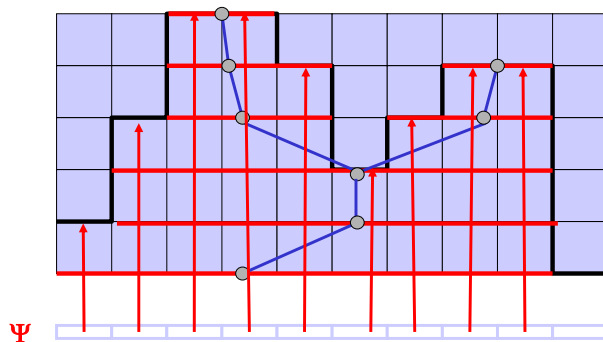
- Any two components of  $F$  are either nested or disjoint

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

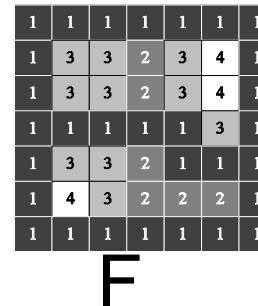


**Components + inclusion relation  $\rightarrow$  tree structure**

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



## Level-sets, components: the 2D case



## 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<sub>1</sub>

## 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	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<sub>2</sub>

## 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<sub>3</sub>

## Level-sets, components: the 2D case

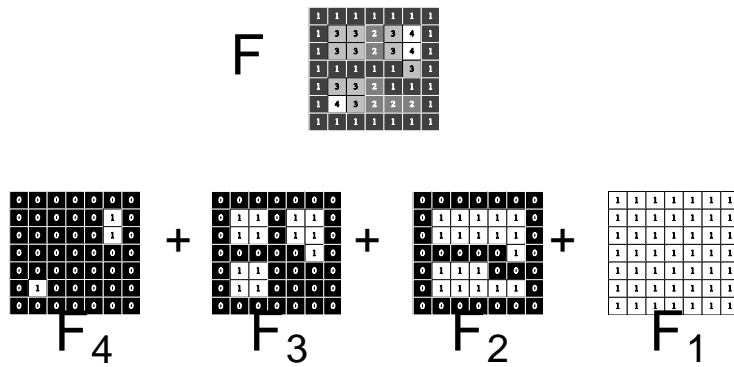
1	1	1	1	1	1	1
1	3	3	2	3	4	1
1	3	3	2	3	4	1
1	1	1	1	1	3	1
1	3	3	2	1	1	1
1	4	3	2	2	2	1
1	1	1	1	1	1	1

F

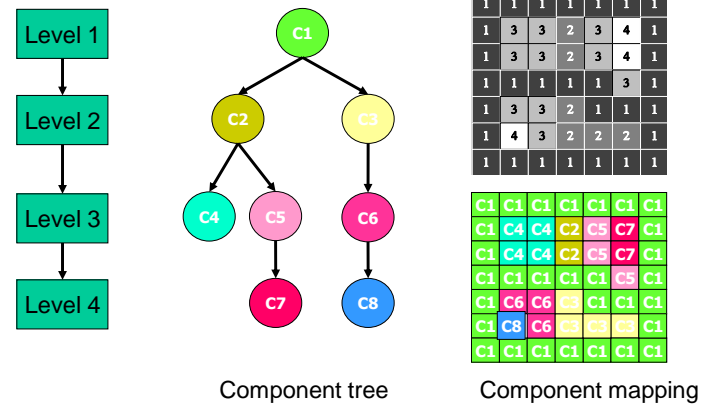
0	0	0	0	0	0	0
0	0	0	0	0	1	0
0	0	0	0	0	1	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	1	0	0	0	0	0
0	0	0	0	0	0	0

F<sub>4</sub>

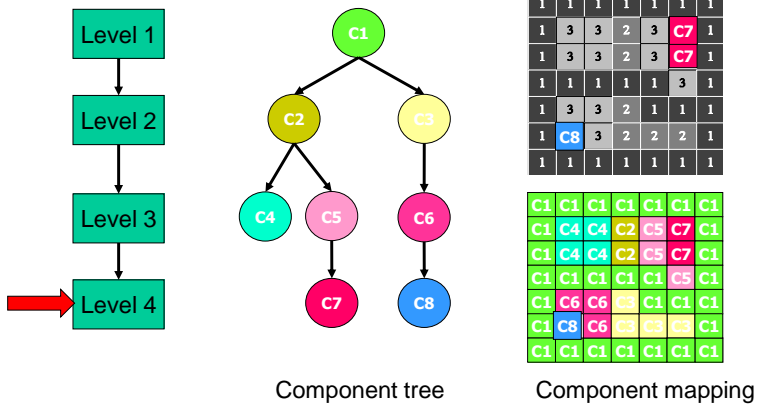
## Cross-sections, components: the 2D case



## Component tree: the 2D case

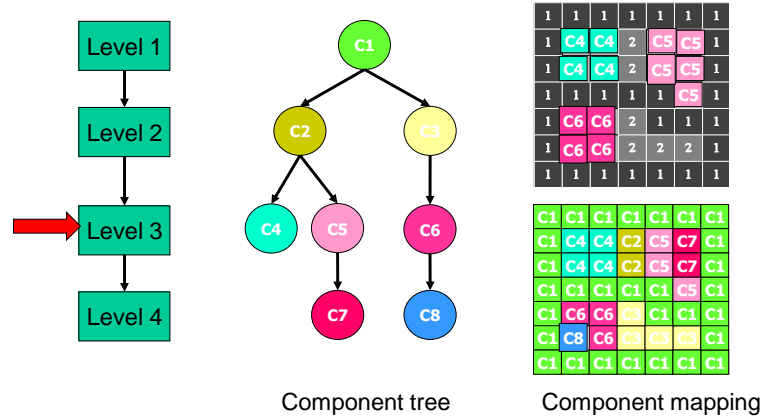


## Component tree: the 2D case



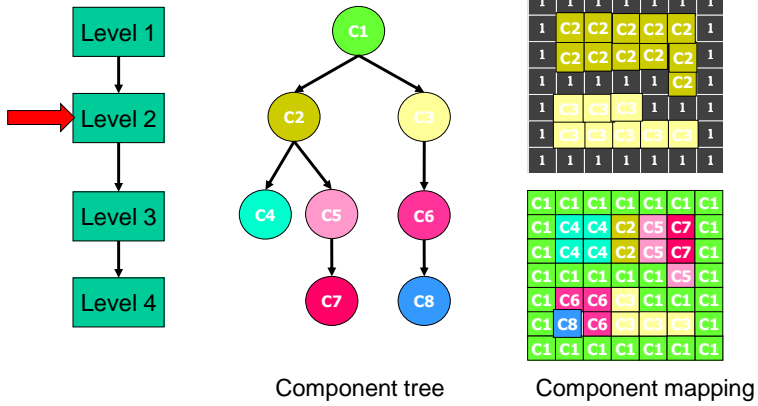
29

## Component tree: the 2D case



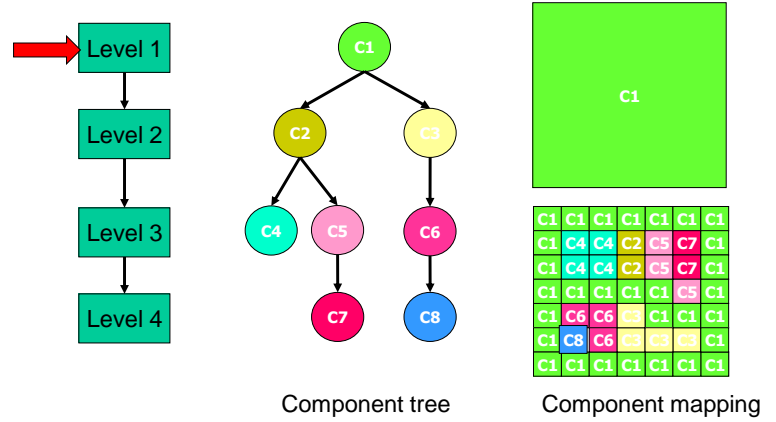
30

## Component tree: the 2D case



31

## Component tree: the 2D case



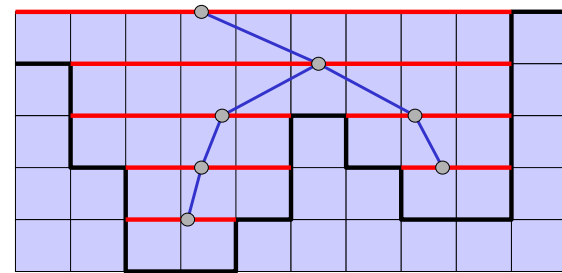
32

## 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

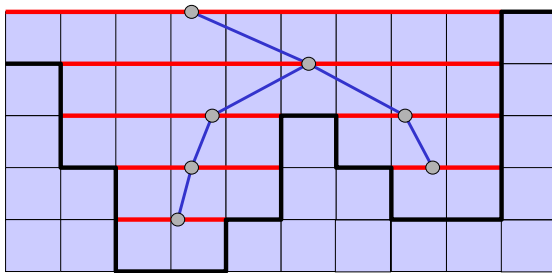
## Flooding: intuitive example

step 1



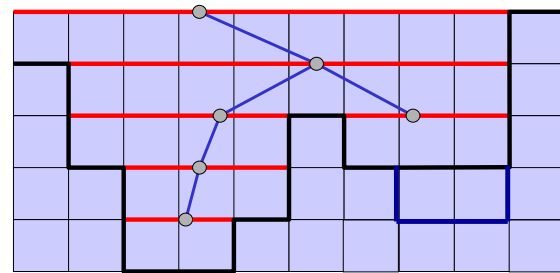
Flooding: intuitive example

step 1



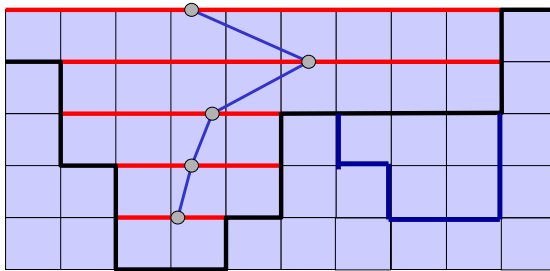
Flooding: intuitive example

step 2



## Flooding: intuitive example

step 3



We remove one branch of  $T$   
We flood one "lobe" of  $F$

## Flooding: formal definition

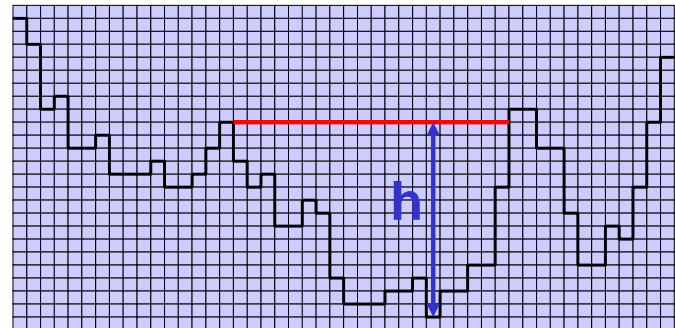
### Definition

- 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  $F$**

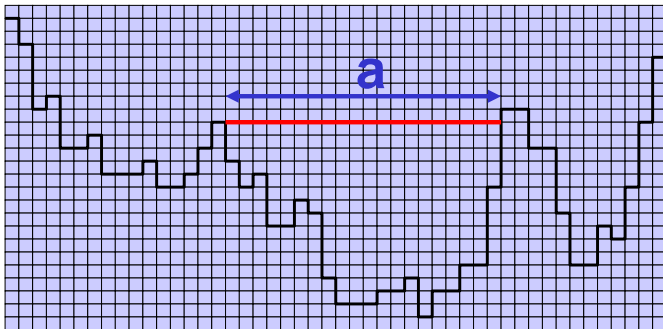
## Filtering a grayscale map

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

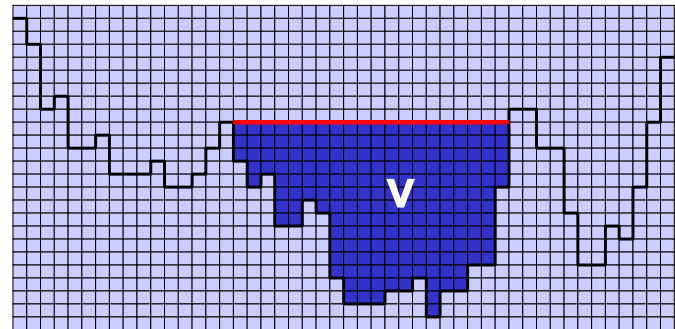
## Attributes: height



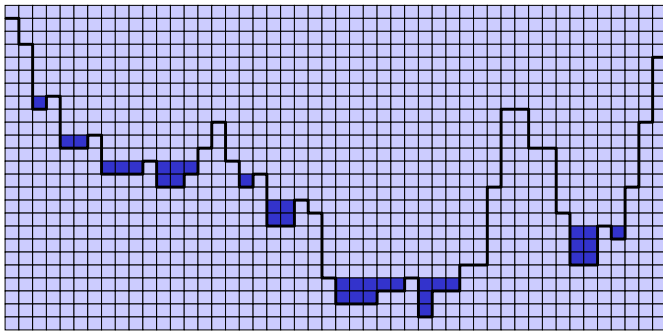
## Attributes: area



## Attributes: volume

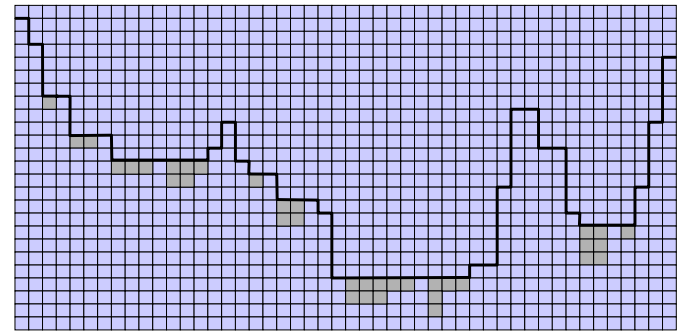


## Filtering by volume



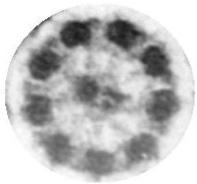
**Regional minima**

## Filtering by volume

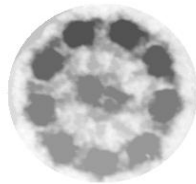
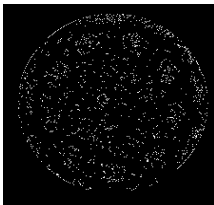


**Regional minima**

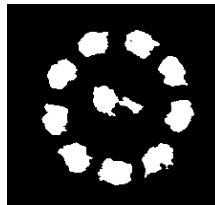
## Application: keep the 10 most volumic lobes



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
  - **Coupric & 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`( $x$ )  $\neq x$ ) 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:**  $O(m \times \alpha(m, |V|))$ 
  - for  $m$  operations
  - $\alpha$ : inverse of Ackerman function (in practice never greater than 4)

## Connected components labeling with union-find

---

**Algorithm 1:** `BuildConnectedComponents`

---

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

- 1 **foreach**  $p \in X$  **do** `MakeSet`( $p$ );
- 2 **foreach**  $p \in X$  **do**
- 3     `compp` := `Find`( $p$ );
- 4     **foreach**  $q \in \Gamma(p)$  *such that*  $q \in X$  **do**
- 5         `compq` := `Find`( $q$ );
- 6         **if** (`compp`  $\neq$  `compq`) **then** `compp` := `Link`(`compq`, `compp`);
- 8 **for**  $p \in X$  **do**  $M(p)$  := `Find`( $p$ );

---

## Connected components labeling with union-find

**Algorithm 1:** BuildConnectedComponents

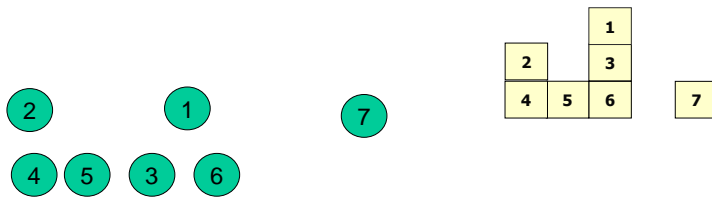
Data:  $(E, \Gamma)$  - graph

Data: A set  $X \subseteq E$

Result:  $M$  - map from  $X$  to  $E$

```

1 foreach  $p \in X$  do MakeSet( $p$ );
2 foreach  $p \in X$  do
3    $comp_p := \text{Find}(p)$ ;
4   foreach  $q \in \Gamma(p)$  such that  $q \in X$  do
5      $comp_q := \text{Find}(q)$ ;
6     if ( $comp_p \neq comp_q$ ) then  $comp_p := \text{Link}(comp_q, comp_p)$ ;
8 for  $p \in X$  do  $M(p) := \text{Find}(p)$ ;
    
```



## Connected components labeling with union-find

**Algorithm 1:** BuildConnectedComponents

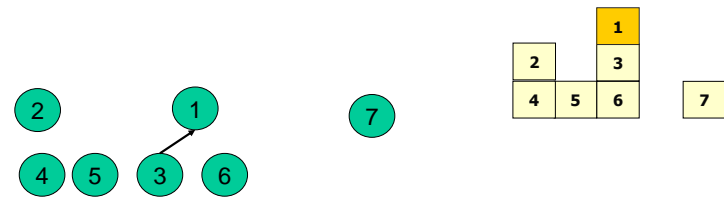
Data:  $(E, \Gamma)$  - graph

Data: A set  $X \subseteq E$

Result:  $M$  - map from  $X$  to  $E$

```

1 foreach  $p \in X$  do MakeSet( $p$ );
2 foreach  $p \in X$  do
3    $comp_p := \text{Find}(p)$ ;
4   foreach  $q \in \Gamma(p)$  such that  $q \in X$  do
5      $comp_q := \text{Find}(q)$ ;
6     if ( $comp_p \neq comp_q$ ) then  $comp_p := \text{Link}(comp_q, comp_p)$ ;
8 for  $p \in X$  do  $M(p) := \text{Find}(p)$ ;
    
```

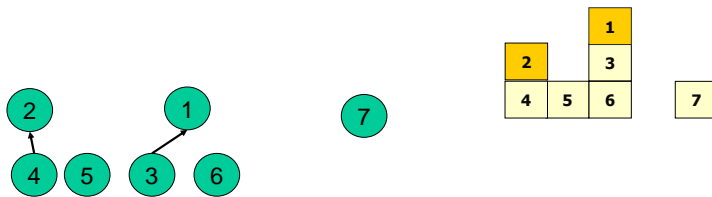


## Connected components labeling with union-find

**Algorithm 1:** BuildConnectedComponents

```

Data:  $(E, \Gamma)$  - graph
Data: A set  $X \subseteq E$ 
Result:  $M$  - map from  $X$  to  $E$ 
1 foreach  $p \in X$  do MakeSet( $p$ );
2 foreach  $p \in X$  do
3   comp $p$  := Find( $p$ );
4   foreach  $q \in \Gamma(p)$  such that  $q \in X$  do
5     comp $q$  := Find( $q$ );
6     if (comp $p$   $\neq$  comp $q$ ) then comp $p$  := Link(comp $q$ , comp $p$ );
8 for  $p \in X$  do  $M(p)$  := Find( $p$ );
    
```

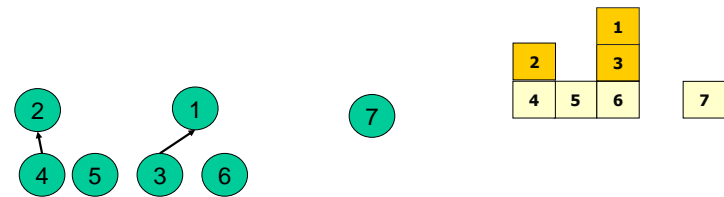


## Connected components labeling with union-find

**Algorithm 1:** BuildConnectedComponents

```

Data:  $(E, \Gamma)$  - graph
Data: A set  $X \subseteq E$ 
Result:  $M$  - map from  $X$  to  $E$ 
1 foreach  $p \in X$  do MakeSet( $p$ );
2 foreach  $p \in X$  do
3   comp $p$  := Find( $p$ );
4   foreach  $q \in \Gamma(p)$  such that  $q \in X$  do
5     comp $q$  := Find( $q$ );
6     if (comp $p$   $\neq$  comp $q$ ) then comp $p$  := Link(comp $q$ , comp $p$ );
8 for  $p \in X$  do  $M(p)$  := Find( $p$ );
    
```



## Connected components labeling with union-find

**Algorithm 1:** BuildConnectedComponents

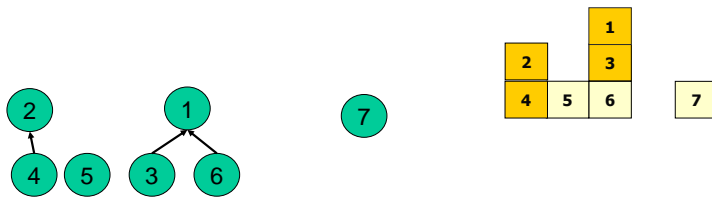
---

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

```

1  foreach  $p \in X$  do MakeSet( $p$ );
2  foreach  $p \in X$  do
3      comp $p$  := Find( $p$ );
4      foreach  $q \in \Gamma(p)$  such that  $q \in X$  do
5          comp $q$  := Find( $q$ );
6          if (comp $p$   $\neq$  comp $q$ ) then comp $p$  := Link(comp $q$ , comp $p$ );
8  for  $p \in X$  do  $M(p)$  := Find( $p$ );
    
```

---



## Connected components labeling with union-find

**Algorithm 1:** BuildConnectedComponents

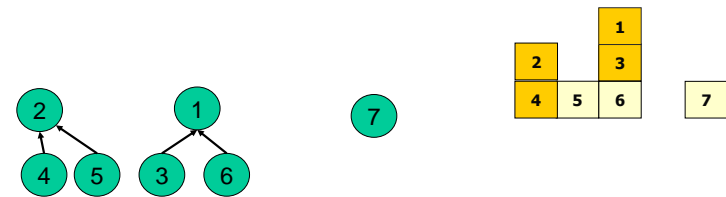
---

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

```

1  foreach  $p \in X$  do MakeSet( $p$ );
2  foreach  $p \in X$  do
3      comp $p$  := Find( $p$ );
4      foreach  $q \in \Gamma(p)$  such that  $q \in X$  do
5          comp $q$  := Find( $q$ );
6          if (comp $p$   $\neq$  comp $q$ ) then comp $p$  := Link(comp $q$ , comp $p$ );
8  for  $p \in X$  do  $M(p)$  := Find( $p$ );
    
```

---



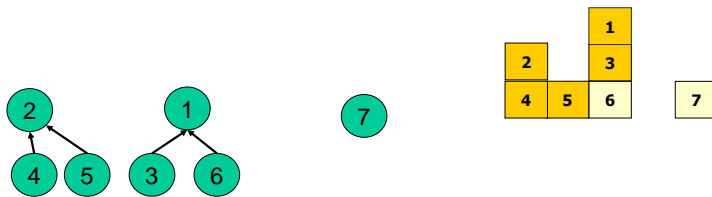
## Connected components labeling with union-find

**Algorithm 1:** BuildConnectedComponents

```

Data:  $(E, \Gamma)$  - graph
Data: A set  $X \subseteq E$ 
Result:  $M$  - map from  $X$  to  $E$ 
1 foreach  $p \in X$  do MakeSet( $p$ );
2 foreach  $p \in X$  do
3   comp $p$  := Find( $p$ );
4   foreach  $q \in \Gamma(p)$  such that  $q \in X$  do
5     comp $q$  := Find( $q$ );
6     if (comp $p$   $\neq$  comp $q$ ) then comp $p$  := Link(comp $q$ , comp $p$ );
8 for  $p \in X$  do  $M(p)$  := Find( $p$ );

```



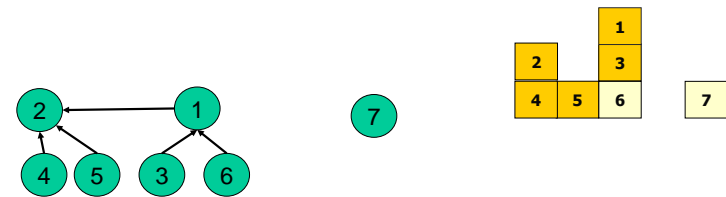
## Connected components labeling with union-find

**Algorithm 1:** BuildConnectedComponents

```

Data:  $(E, \Gamma)$  - graph
Data: A set  $X \subseteq E$ 
Result:  $M$  - map from  $X$  to  $E$ 
1 foreach  $p \in X$  do MakeSet( $p$ );
2 foreach  $p \in X$  do
3   comp $p$  := Find( $p$ );
4   foreach  $q \in \Gamma(p)$  such that  $q \in X$  do
5     comp $q$  := Find( $q$ );
6     if (comp $p$   $\neq$  comp $q$ ) then comp $p$  := Link(comp $q$ , comp $p$ );
8 for  $p \in X$  do  $M(p)$  := Find( $p$ );

```



## Connected components labeling with union-find

**Algorithm 1:** BuildConnectedComponents

Data:  $(E, \Gamma)$  - graph

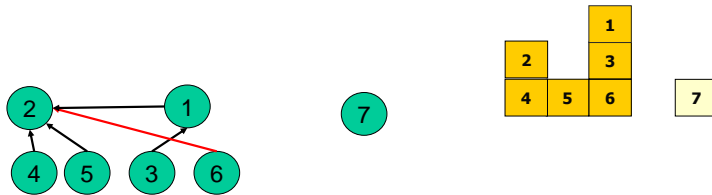
Data: A set  $X \subseteq E$

Result:  $M$  - map from  $X$  to  $E$

```

1 foreach  $p \in X$  do MakeSet( $p$ );
2 foreach  $p \in X$  do
3    $comp_p := \text{Find}(p)$ ;
4   foreach  $q \in \Gamma(p)$  such that  $q \in X$  do
5      $comp_q := \text{Find}(q)$ ;
6     if ( $comp_p \neq comp_q$ ) then  $comp_p := \text{Link}(comp_q, comp_p)$ ;
8 for  $p \in X$  do  $M(p) := \text{Find}(p)$ ;

```



## Connected components labeling with union-find

**Algorithm 1:** BuildConnectedComponents

Data:  $(E, \Gamma)$  - graph

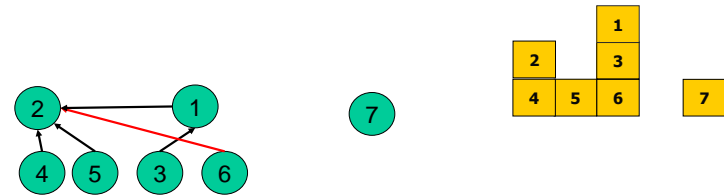
Data: A set  $X \subseteq E$

Result:  $M$  - map from  $X$  to  $E$

```

1 foreach  $p \in X$  do MakeSet( $p$ );
2 foreach  $p \in X$  do
3    $comp_p := \text{Find}(p)$ ;
4   foreach  $q \in \Gamma(p)$  such that  $q \in X$  do
5      $comp_q := \text{Find}(q)$ ;
6     if ( $comp_p \neq comp_q$ ) then  $comp_p := \text{Link}(comp_q, comp_p)$ ;
8 for  $p \in X$  do  $M(p) := \text{Find}(p)$ ;

```



## Connected components labeling with union-find

**Algorithm 1:** BuildConnectedComponents

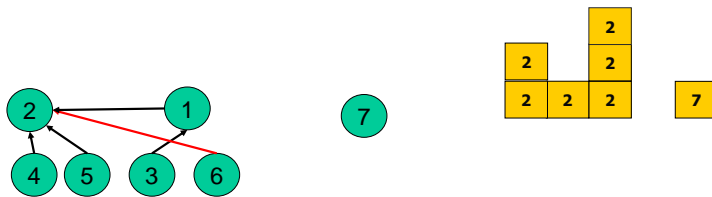
---

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

```

1  foreach  $p \in X$  do MakeSet( $p$ );
2  foreach  $p \in X$  do
3  |   comp $p$  := Find( $p$ );
4  |   foreach  $q \in \Gamma(p)$  such that  $q \in X$  do
5  |   |   comp $q$  := Find( $q$ );
6  |   |   if (comp $p$   $\neq$  comp $q$ ) then comp $p$  := Link(comp $q$ , comp $p$ );
8  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  $(\text{Find}(x) \neq \text{Find}(y))$  **do**
      - $\text{adjNode} := \text{lowestNode}[\text{Find}(y)]$
      - $\text{nodes}[x].\text{addChild}(\text{nodes}[\text{adjNode}])$
      - $\text{curNode} := \text{Link}(x, \text{adjNode})$
      - $\text{lowestNode}[\text{curNode}] := \text{nodes}[x]$

## 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  $(\text{Find}(x) \neq \text{Find}(y))$  **do**
      - $\text{adjNode} := \text{lowestNode}[\text{Find}(y)]$
      - $\text{nodes}[x].\text{addChild}(\text{nodes}[\text{adjNode}])$
      - $\text{curNode} := \text{Link}(x, \text{adjNode})$
      - $\text{lowestNode}[\text{curNode}] := \text{nodes}[x]$

Image

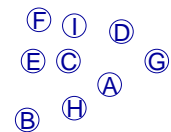
A:7	B:1	C:9
D:6	E:4	F:2
G:8	H:5	I:3

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  $(\text{Find}(x) \neq \text{Find}(y))$  **do**
      - $\text{adjNode} := \text{lowestNode}[\text{Find}(y)]$
      - $\text{nodes}[x].\text{addChild}(\text{nodes}[\text{adjNode}])$
      - $\text{curNode} := \text{Link}(x, \text{adjNode})$
      - $\text{lowestNode}[\text{curNode}] := \text{nodes}[x]$

$x = C$

Image

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

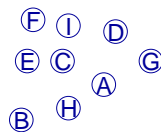
Lowest Node

A	B	C
D	E	F
G	H	I

nodes

Ⓒ

Disjoint sets



## 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  $(\text{Find}(x) \neq \text{Find}(y))$  **do**
      - $\text{adjNode} := \text{lowestNode}[\text{Find}(y)]$
      - $\text{nodes}[x].\text{addChild}(\text{nodes}[\text{adjNode}])$
      - $\text{curNode} := \text{Link}(x, \text{adjNode})$
      - $\text{lowestNode}[\text{curNode}] := \text{nodes}[x]$

$x = G$

Image

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

Lowest Node

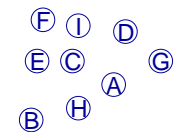
A	B	C
D	E	F
G	H	I

nodes

Ⓒ

Ⓒ

Disjoint sets



### 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  $(\text{Find}(x) \neq \text{Find}(y))$  **do**
      - $\text{adjNode} := \text{lowestNode}[\text{Find}(y)]$
      - $\text{nodes}[x].\text{addChild}(\text{nodes}[\text{adjNode}])$
      - $\text{curNode} := \text{Link}(x, \text{adjNode})$
      - $\text{lowestNode}[\text{curNode}] := \text{nodes}[x]$

$x = A$

Image

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

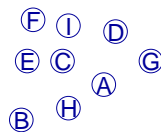
Lowest Node

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  $(\text{Find}(x) \neq \text{Find}(y))$  **do**
      - $\text{adjNode} := \text{lowestNode}[\text{Find}(y)]$
      - $\text{nodes}[x].\text{addChild}(\text{nodes}[\text{adjNode}])$
      - $\text{curNode} := \text{Link}(x, \text{adjNode})$
      - $\text{lowestNode}[\text{curNode}] := \text{nodes}[x]$

$x = D$

Image

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

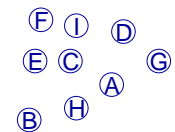
Lowest Node

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  $(\text{Find}(x) \neq \text{Find}(y))$  **do**
      - $\text{adjNode} := \text{lowestNode}[\text{Find}(y)]$
      - $\text{nodes}[x].\text{addChild}(\text{nodes}[\text{adjNode}])$
      - $\text{curNode} := \text{Link}(x, \text{adjNode})$
      - $\text{lowestNode}[\text{curNode}] := \text{nodes}[x]$

$x = D$   
 $y = A$

Image

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

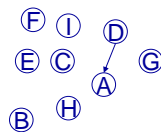
Lowest Node

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  $(\text{Find}(x) \neq \text{Find}(y))$  **do**
      - $\text{adjNode} := \text{lowestNode}[\text{Find}(y)]$
      - $\text{nodes}[x].\text{addChild}(\text{nodes}[\text{adjNode}])$
      - $\text{curNode} := \text{Link}(x, \text{adjNode})$
      - $\text{lowestNode}[\text{curNode}] := \text{nodes}[x]$

$x = D$   
 $y = G$

Image

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

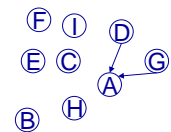
Lowest Node

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  $(\text{Find}(x) \neq \text{Find}(y))$  **do**
      - $\text{adjNode} := \text{lowestNode}[\text{Find}(y)]$
      - $\text{nodes}[x].\text{addChild}(\text{nodes}[\text{adjNode}])$
      - $\text{curNode} := \text{Link}(x, \text{adjNode})$
      - $\text{lowestNode}[\text{curNode}] := \text{nodes}[x]$

$x = H$   
 $y = G$

Image

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

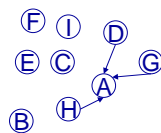
Lowest Node

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  $(\text{Find}(x) \neq \text{Find}(y))$  **do**
      - $\text{adjNode} := \text{lowestNode}[\text{Find}(y)]$
      - $\text{nodes}[x].\text{addChild}(\text{nodes}[\text{adjNode}])$
      - $\text{curNode} := \text{Link}(x, \text{adjNode})$
      - $\text{lowestNode}[\text{curNode}] := \text{nodes}[x]$

$x = E$   
 $y = D$

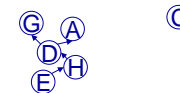
Image

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

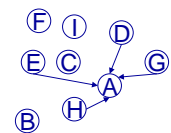
Lowest Node

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  $(\text{Find}(x) \neq \text{Find}(y))$  **do**
      - $\text{adjNode} := \text{lowestNode}[\text{Find}(y)]$
      - $\text{nodes}[x].\text{addChild}(\text{nodes}[\text{adjNode}])$
      - $\text{curNode} := \text{Link}(x, \text{adjNode})$
      - $\text{lowestNode}[\text{curNode}] := \text{nodes}[x]$

$x = E$   
 $y = H$

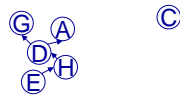
Image

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

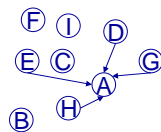
Lowest Node

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  $(\text{Find}(x) \neq \text{Find}(y))$  **do**
      - $\text{adjNode} := \text{lowestNode}[\text{Find}(y)]$
      - $\text{nodes}[x].\text{addChild}(\text{nodes}[\text{adjNode}])$
      - $\text{curNode} := \text{Link}(x, \text{adjNode})$
      - $\text{lowestNode}[\text{curNode}] := \text{nodes}[x]$

$x = I$   
 $y = H$

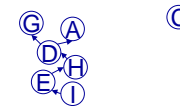
Image

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

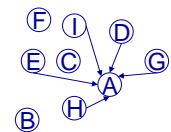
Lowest Node

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  $(\text{Find}(x) \neq \text{Find}(y))$  **do**
      - $\text{adjNode} := \text{lowestNode}[\text{Find}(y)]$
      - $\text{nodes}[x].\text{addChild}(\text{nodes}[\text{adjNode}])$
      - $\text{curNode} := \text{Link}(x, \text{adjNode})$
      - $\text{lowestNode}[\text{curNode}] := \text{nodes}[x]$

$x = F$   
 $y = C$

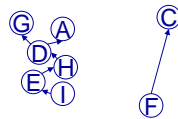
Image

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

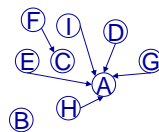
Lowest Node

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  $(\text{Find}(x) \neq \text{Find}(y))$  **do**
      - $\text{adjNode} := \text{lowestNode}[\text{Find}(y)]$
      - $\text{nodes}[x].\text{addChild}(\text{nodes}[\text{adjNode}])$
      - $\text{curNode} := \text{Link}(x, \text{adjNode})$
      - $\text{lowestNode}[\text{curNode}] := \text{nodes}[x]$

$x = F$   
 $y = E$

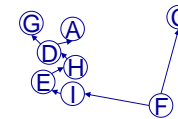
Image

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

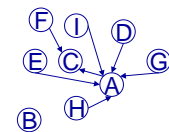
Lowest Node

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  $(\text{Find}(x) \neq \text{Find}(y))$  **do**
      - $\text{adjNode} := \text{lowestNode}[\text{Find}(y)]$
      - $\text{nodes}[x].\text{addChild}(\text{nodes}[\text{adjNode}])$
      - $\text{curNode} := \text{Link}(x, \text{adjNode})$
      - $\text{lowestNode}[\text{curNode}] := \text{nodes}[x]$

$x = F$   
 $y = I$

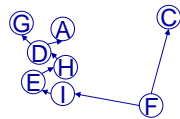
Image

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

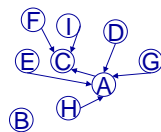
Lowest Node

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  $(\text{Find}(x) \neq \text{Find}(y))$  **do**
      - $\text{adjNode} := \text{lowestNode}[\text{Find}(y)]$
      - $\text{nodes}[x].\text{addChild}(\text{nodes}[\text{adjNode}])$
      - $\text{curNode} := \text{Link}(x, \text{adjNode})$
      - $\text{lowestNode}[\text{curNode}] := \text{nodes}[x]$

$x = B$   
 $y = A$

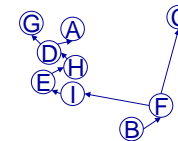
Image

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

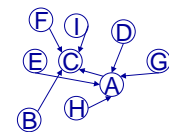
Lowest Node

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  $(\text{Find}(x) \neq \text{Find}(y))$  **do**
      - $\text{adjNode} := \text{lowestNode}[\text{Find}(y)]$
      - $\text{nodes}[x].\text{addChild}(\text{nodes}[\text{adjNode}])$
      - $\text{curNode} := \text{Link}(x, \text{adjNode})$
      - $\text{lowestNode}[\text{curNode}] := \text{nodes}[x]$

$x = B$   
 $y = C$

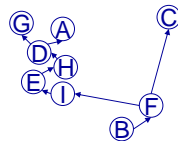
Image

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

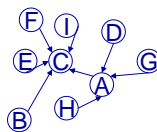
Lowest Node

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  $(\text{Find}(x) \neq \text{Find}(y))$  **do**  $O(|E| \cdot \alpha(|E|, |V|))$ 
      - $\text{adjNode} := \text{lowestNode}[\text{Find}(y)]$   $O(|E|)$
      - $\text{nodes}[x].\text{addChild}(\text{nodes}[\text{adjNode}])$   $O(|E|)$
      - $\text{curNode} := \text{Link}(x, \text{adjNode})$   $O(|E| \cdot \alpha(|E|, |V|))$
      - $\text{lowestNode}[\text{curNode}] := \text{nodes}[x]$   $O(|E|)$

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

```

1  $N_n := N$ ; Sort the points in decreasing order of level for  $F$ ;
2 foreach  $p \in E$  do {MakeSet1( $p$ ); MakeSet2( $p$ );  $nodes[p] := MakeNode(F(p))$ ;  $subtreeRoot[p] := p$ };
3 foreach  $p \in E$  in decreasing order of level for  $F$  do
4    $curCanonicalElt := Find1(p)$ ;
5    $curNode := Find2(subtreeRoot[curCanonicalElt])$ ;
6   foreach already processed neighbor  $q$  of  $p$  with  $F(q) \geq F(p)$  do
7      $adjCanonicalElt := Find1(q)$ ;
8      $adjNode := Find2(subtreeRoot[adjCanonicalElt])$ ;
9     if ( $curNode \neq adjNode$ ) then
10      if ( $nodes[curNode] \rightarrow level == nodes[adjNode] \rightarrow level$ ) then
11         $curNode := MergeNodes(adjNode, curNode)$ ;
12      else
13        //We have  $nodes[curNode] \rightarrow level < nodes[adjNode] \rightarrow level$ 
14         $nodes[curNode] \rightarrow addSon(nodes[adjNode])$ ;
15         $curCanonicalElt := Link1(adjCanonicalElt, curCanonicalElt)$ ;
16         $subtreeRoot[curCanonicalElt] := curNode$ ;
17
18  $Root := subtreeRoot[Find1(Find2(0))]$ ;
19 foreach  $p \in E$  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 node1, int node2*)  
 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<sub>n</sub> := N<sub>n</sub> - 1;  
 return tmpNode;

---

## Example

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

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

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

[12] 120

## Step 2

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

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

Fth1

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

Fth2

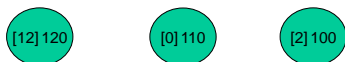
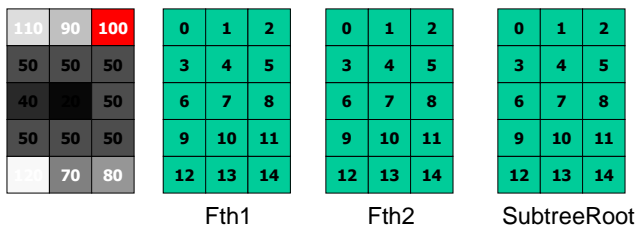
0	1	2
3	4	5
6	7	8
9	10	11
12	13	14

SubtreeRoot

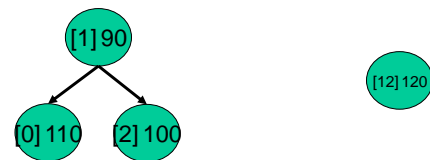
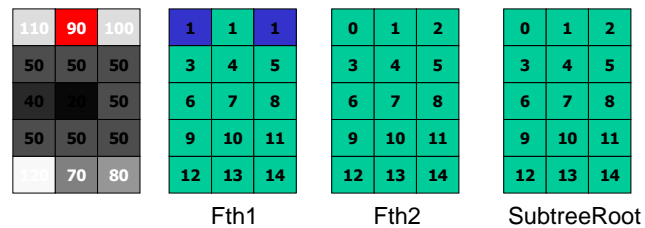
[12] 120

[0] 110

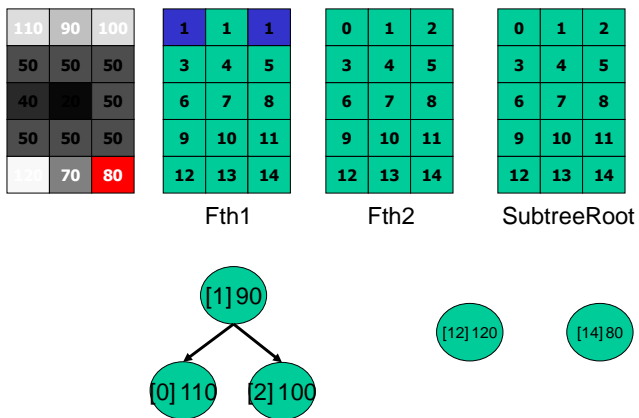
### step 3



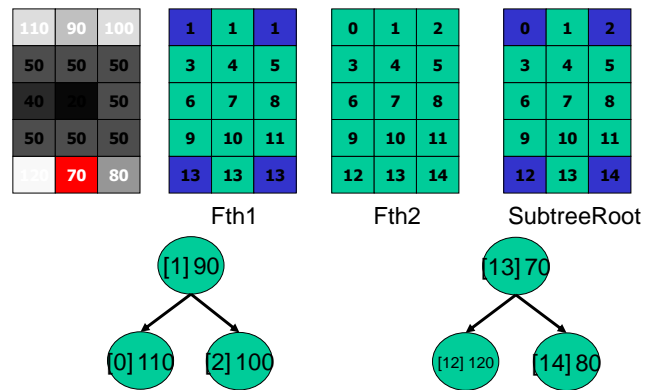
### step 4



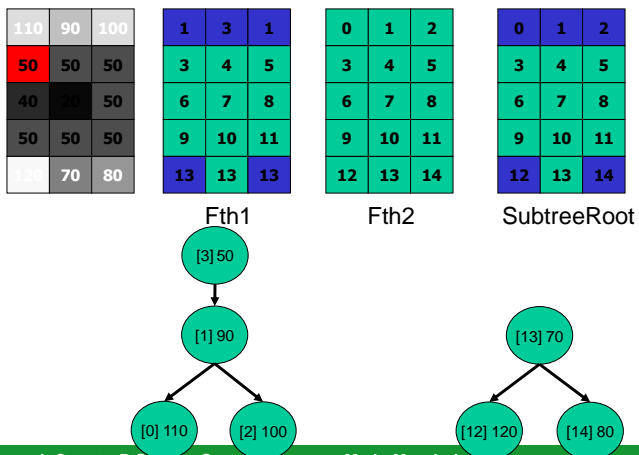
### step 5



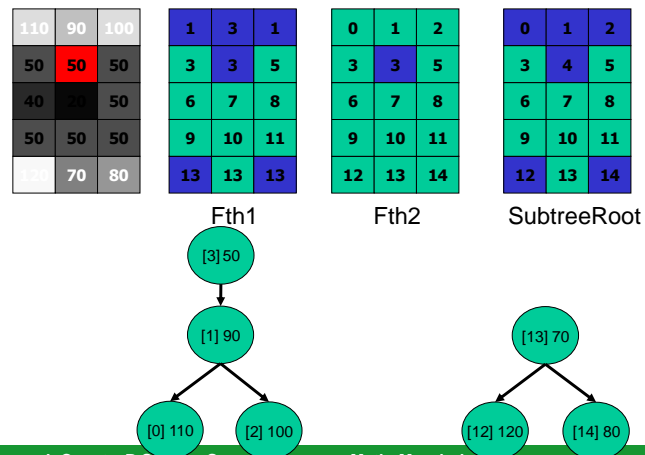
### step 6



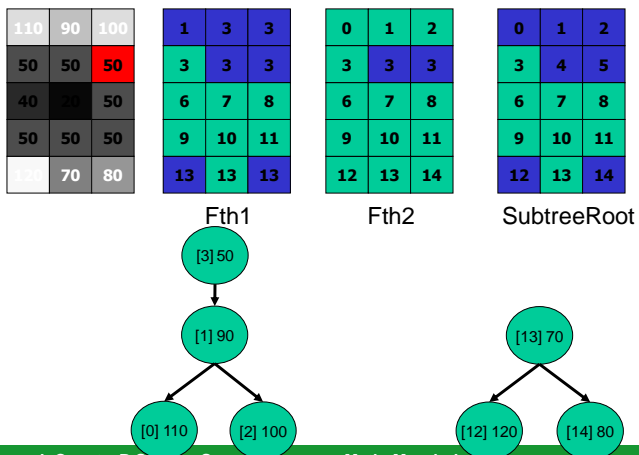
### step 7



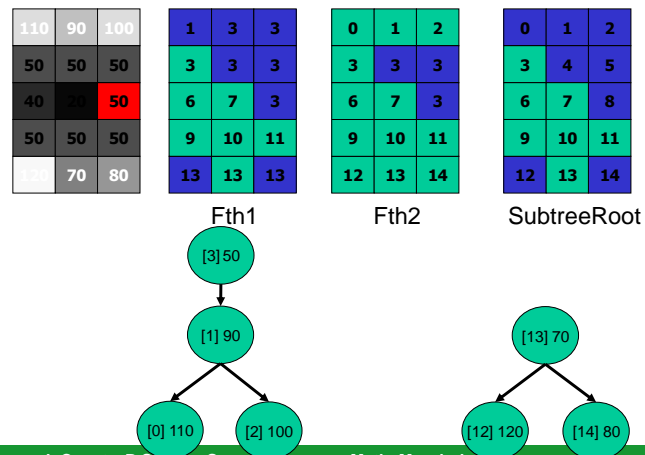
### step 8



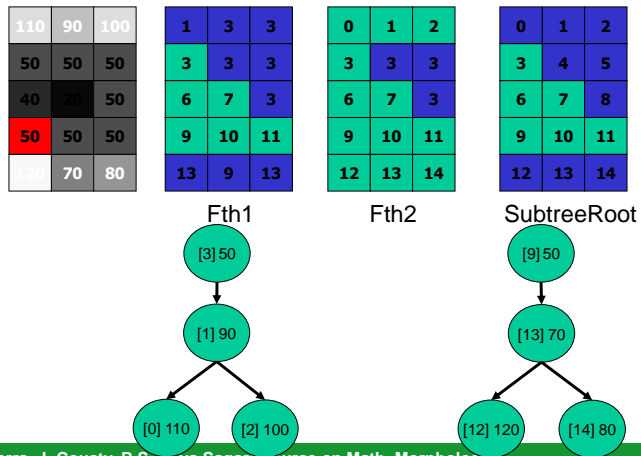
### step 9



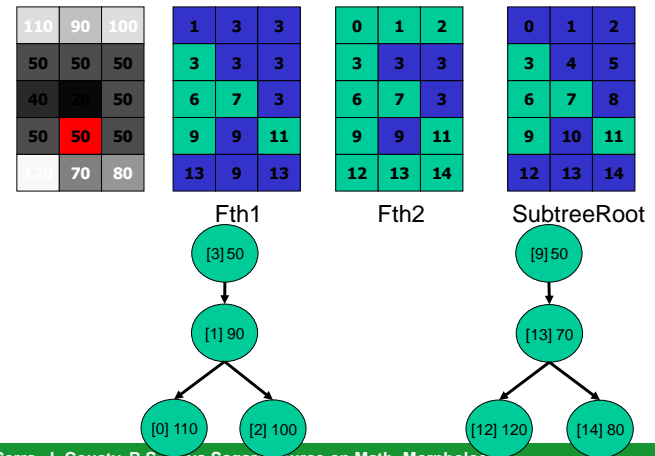
### step 10



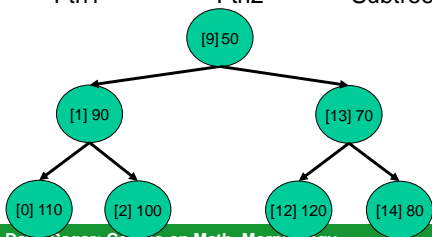
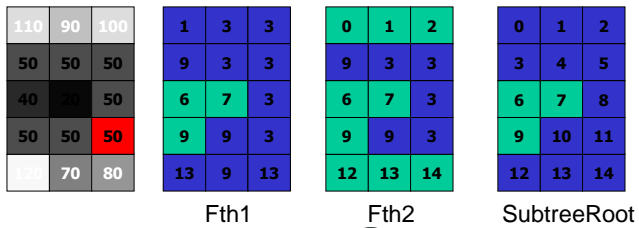
## step 11



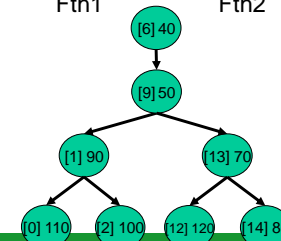
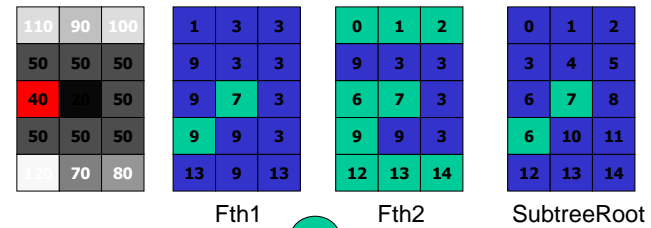
## step 12



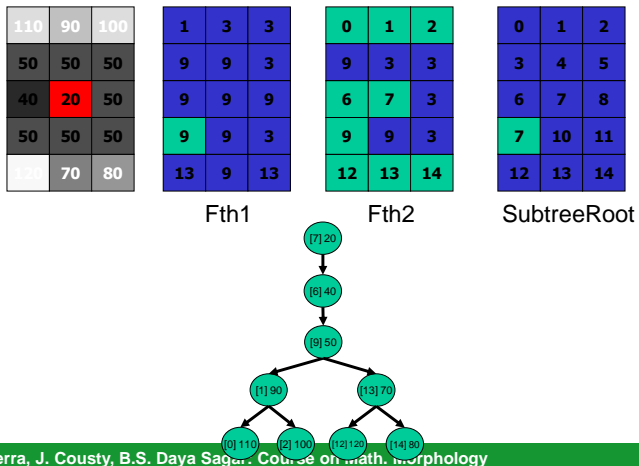
### step 13



### step 14



## step 15



## 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