Building the component tree in quasi-linear time

L. Najman and M. Couprie
Institut Gaspard-Monge
Groupe ESIEE, Laboratoire A2SI
BP99, 93162 Noisy-le-Grand Cedex France
{l.najman,m.couprie}@esiee.fr

***Abstract*— The level sets of a map are the sets of points with level above a given threshold. The connected components of the level sets, thanks to the inclusion relation, can be organized in a tree structure, that is called the *component tree*. This tree, under several variations, has been used in numerous applications. Various algorithms have been proposed in the literature for computing the component tree. The fastest ones (considering the worst-case complexity) have been proved to run in $\mathbf{O}(n \ln(n))$. In this paper, we propose a simple to implement quasi-linear algorithm for computing the component tree on symmetric graphs, based on Tarjan's union-find procedure. We also propose an algorithm that computes the n most significant lobes of a map.**

***Index Terms*— Component tree, connected operators, mathematical morphology, classification, disjoint sets, union-find, image and signal processing, filtering**

I. Introduction

The level sets of a map are the sets of points with level above a given threshold. The connected components of the level sets, thanks to the inclusion relation, can be organized in a tree structure, that is called the *component tree*. The component tree captures some essential features of the map. It has been used (under several variations) in numerous applications among which we can cite: image filtering and segmentation [12], [11], [7], [14], video segmentation [21], image registration [16], [18], image compression [21] and data visualization [5]. This tree is also fundamental for the efficient computation of the topological watershed introduced by M. Couprie and G. Bertrand [7], [8], [3].

While having been (re)discovered by several authors for image processing applications, the component tree concept was first introduced in statistics [26], [13] for classification and clustering. For image processing, the use of this tree in order to represent the "meaningful" information contained in a numerical function can be found in particular, in a paper by Hanusse and Guillataud [12], [11]; the authors claim that this tree can play a central role in image segmentation, and suggest a way to compute it, based on an immersion simulation. Several authors, such as Vachier [25], Breen and Jones [4], Salembier et al. [21] have used some variations of this structure in order to implement efficiently some morphological operators (e.g. connected operators [22], granulometries, extinction functions, dynamics [2]).

Let us describe informally an "emergence" process that will later help us designing an algorithm for building the component tree. Using topographical references, we see the map as the surface of a relief, with the level of a point corresponding to its altitude. Imagine that the surface is completely covered by water, and that the level of water slowly decreases. Islands (regional maxima) appear. These islands form the leafs of the component tree. As the level of water decreases, islands grow, building the branches of the tree. Sometimes, at a given level, several islands merge into one connected piece. Such pieces are the forks of the tree. We stop when all the water has disappeared. The emerged area forms a unique component: the root of the tree.

Various algorithms have been proposed in the literature for computing the component tree [4], [21], [15], the latter reference also contains a discussion about time complexity of the different algorithms. The fastest ones (considering the worst-case complexity) have been proved to run in $\mathrm{O}(n \ln(n))$, where n denotes the number of pixels of the image. In this paper[1], we propose a quasi-linear algorithm for computing the component tree of functions defined on general symmetric graphs, based on Tarjan's union-find [24] procedure. More precisely, our algorithm runs in $O(N \times \alpha(N))$ where N denotes the size of the graph (number of vertices + number of edges) and α is a very slow-growing "diagonal inverse" of the Ackermann's function (we have $\alpha(10^{80}) \approx 4$). We would like to emphasize that this algorithm is simple to implement.

The paper is organised as follows: we first recall the definitions of some basic graph notions and define the component tree in this framework. We explain the disjoint set problem, together with the solution proposed by Tarjan. Using a disjoint set fomulation, we present our component tree algorithm, and we describe its execution on an example. We then show that the proposed algorithm is quasi-linear with respect to the size of the graph, and compare it to one of the most cited component tree algorithm. We illustrate the use of the component tree for automatic detection of some image features, based on a unique parameter which is the number of features that we expect to find in the image.

II. Vertex-weighted graph and component tree

A. Basic notions for graphs

Let V be a finite set of vertices (or points), and let $\mathcal{P}(V)$ denote the set of all subsets of V. Throughout this paper, E denotes a binary relation on V (that is, a subset of the cartesian product $V \times V$) which is anti-reflexive ($(x, x) \notin E$) and symmetric ($(x, y) \in E \Leftrightarrow (y, x) \in E$). We say that the

[1]A preliminary and reduced version of this paper appeared in conference proceedings as [19]. This work has been partially supported by the CNRS.

pair (V, E) is a *graph*, and the elements of E are called *edges*. We denote by Γ the map from V to $\mathcal{P}(V)$ such that, for all $x \in V$, $\Gamma(x) = \{y \in V | (x, y) \in E\}$. For any point x, the set $\Gamma(x)$ is called the *neighborhood of* x. If $y \in \Gamma(x)$ then we say that y is a *neighbor* of x and that x and y are *adjacent*.

Let $X \subseteq V$. Let $x_0, x_n \in X$. A *path from* x_0 *to* x_n *in* X is a sequence $\pi = (x_0, x_1, \ldots, x_n)$ of points of X such that $x_{i+1} \in \Gamma(x_i)$, with $i = 0 \ldots n-1$. Let $x, y \in X$, we say that x and y are *linked for* X if there exists a path from x to y in X. We say that X is *connected* if any x and y in X are linked for X. We say that $Y \subseteq V$ is a *connected component of* X if $Y \subseteq X$, Y is connected, and Y is maximal for these two properties (*i.e.*, $Y = Z$ whenever $Y \subseteq Z \subseteq X$ and Z is connected).

In the following, we assume that the graph (V, E) is connected, that is, V is made of exactly one connected component.

B. Basic notions for vertex-weighted graphs

We denote by $\mathcal{F}(V, D)$, or simply by $\mathcal{F}$, the set composed of all maps from V to D, where D can be any finite set equipped with a total order (*e.g.,* a finite subset of the set of rational numbers or of the set of integers). For a map $F \in \mathcal{F}$, the triplet (V, E, F) is called a *(vertex-)weighted graph*. For a point $p \in V$, $F(p)$ is called the *weight* or *level* of p.

Let $F \in \mathcal{F}$, we define $F_k = \{x \in V | F(x) \geq k\}$ with $k \in D$; F_k is called a *(cross-)section* of F. A connected component of a section F_k is called a *(level k) component* of F. A level k component of F that does not contain a level $(k+1)$ component of F is called a *(regional) maximum* of F. We define $k_{\min} = \min \{F(x) | x \in V\}$ and $k_{\max} = \max \{F(x) | x \in V\}$, which represent respectively, the minimum and the maximum level in the map F.

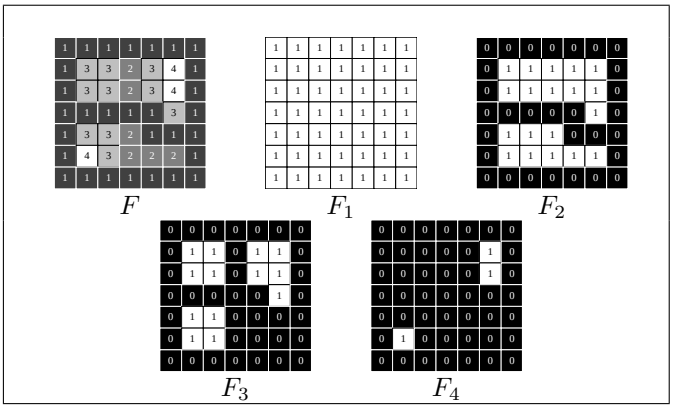
Although the notions we are dealing with in this paper are defined for general graphs, we are going to illustrate our work with the case of 2D images that we model by weighted graphs. Let $\mathbb{Z}$ denote the set of integers. We choose for V a subset of $\mathbb{Z}^2$. A point $x \in V$ is defined by its two coordinates (x_1, x_2). We choose for E the 4-connected adjacency relation defined by $E = \{(x, y) \in V \times V \ ; \ |x_1 - y_1| + |x_2 - y_2| = 1\}$.

Fig. 1.a shows a weighted graph (V, E, F) and four cross-sections of F, between the level $k_{\min} = 1$ and the level $k_{\max} = 4$. The set F_4 is made of two connected components which are regional maxima of F.

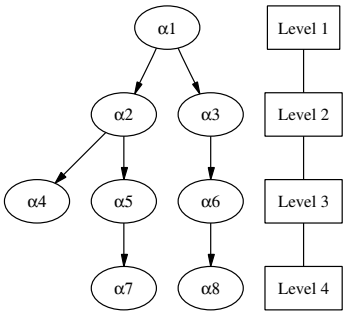
C. Component Tree

From the example of Fig. 1.a, we can see that the connected components of the different cross-sections may be organized, thanks to the inclusion relation, to form a tree structure (see also [2]).

Let $F \in \mathcal{F}$. For any component c of F, we set $h(c) = \max\{k | c \text{ is a level } k \text{ component of } F\}$. Note that $h(c) = \min\{F(x) | x \in c\}$. We define $\mathcal{C}(F)$ as the set composed of all the pairs $[k, c]$, where c is a component of F and $k = h(c)$. We call *altitude of* $[k, c]$ the number k. Remark that $[k_1, c] \in \mathcal{C}(F)$ and $[k_2, c] \in \mathcal{C}(F)$ implies $k_1 = k_2$, in other words, any two distinct elements of $\mathcal{C}(F)$ correspond to distinct sets of points.



(a)



(b)

α_1	α_1	α_1	α_1	α_1	α_1	α_1
α_1	α_4	α_4	α_2	α_5	α_7	α_1
α_1	α_4	α_4	α_2	α_5	α_7	α_1
α_1	α_1	α_1	α_1	α_1	α_5	α_1
α_1	α_6	α_6	α_3	α_1	α_1	α_1
α_1	α_8	α_6	α_3	α_3	α_3	α_1
α_1	α_1	α_1	α_1	α_1	α_1	α_1

(c)

Fig. 1. (a) A vertex-weighted graph (V, E, F) and its cross-sections at levels 1, 2, 3, 4. (b) The component tree of F. (c) The associated component mapping. The component at level 1 is called α_1, the two components at level 2 are called α_2 and α_3 (according to the usual scanning order), and so on.

Let $F \in \mathcal{F}$, let $[k_1, c_1], [k_2, c_2]$ be distinct elements of $\mathcal{C}(F)$. We say that $[k_1, c_1]$ is the *parent of* $[k_2, c_2]$ if $c_2 \subset c_1$ and if there is no other $[k_3, c_3]$ in $\mathcal{C}(F)$ such that $c_1 \subset c_3 \subset c_2$. In this case we also say that $[k_2, c_2]$ is a *child of* $[k_1, c_1]$. With this relation "parent", $\mathcal{C}(F)$ forms a directed tree that we call the *component tree of* F, and that we will also denote by $\mathcal{C}(F)$ by abuse of terminology. Any element of $\mathcal{C}(F)$ is called a *node*. An element of $\mathcal{C}(F)$ which has no child (a maximum of F) is called a *leaf*, the node which has no parent (*i.e.*, $[k_{\min}, V]$) is called the *root*.

We define the *component mapping* M as the map which associates to each point $p \in V$ the node $[k, c]$ of $\mathcal{C}(F)$ such that $p \in c$ and $F(p) = k$. The component mapping is necessary for using the component tree in applications.

Fig. 1.b shows the component tree of the weighted graph depicted in Fig. 1.a, and Fig. 1.c shows the associated component mapping. The component at level 1 is called α_1, the two components at level 2 are called α_2 and α_3 (according to the usual scanning order), and so on.

III. Component tree quasi-linear algorithm

A. Disjoint Sets

The disjoint set problem consists in maintaining a collection $\mathcal{Q}$ of disjoint subsets of a set V under the operation of union. Each set X in $\mathcal{Q}$ is represented by a unique element of X, called the *canonical element*. In the following, x and y denote two distinct elements of V. The collection is managed by three operations:

- **MakeSet**(x): add the set $\{x\}$ to the collection $\mathcal{Q}$, provided that the element x does not already belongs to a set in $\mathcal{Q}$.
- **Find**(x): return the canonical element of the set in $\mathcal{Q}$ which contains x.
- **Link**(x, y): let X and Y be the two sets in $\mathcal{Q}$ whose canonical elements are x and y respectively (x and y must be different). Both sets are removed from $\mathcal{Q}$, their union $Z = X \cup Y$ is added to $\mathcal{Q}$ and a canonical element for Z is selected and returned.

Tarjan [24] proposed a very simple and very efficient algorithm called *union-find* to achieve any intermixed sequence of such operations with a quasi-linear complexity. More precisely, if m denotes the number of operations and n denotes the number of elements, the worst-case complexity is $O(m \times \alpha(m, n))$ where $\alpha(m, n)$ is a function which grows very slowly, for all practical purposes $\alpha(m, n)$ is never greater than four[2].

The implementation of this algorithm is given below in procedure **MakeSet** and functions **Link** and **Find**. Each set of the collection is represented by a rooted tree, where the canonical element of the set is the root of the tree. To each element x is associated a parent $\text{Par}(x)$ (which is an element) and a rank $\text{Rnk}(x)$ (which is an integer). The mappings 'Par' and 'Rnk' are represented by global arrays in memory. One of the two key heuristics to reduce the complexity is a technique called *path compression*, that is aimed at reducing, in the long run, the cost of **Find**. It consists, after finding the root r of the tree which contains x, in considering each element y of the parent path from x to r (including x), and setting the parent of y to be r. The other key technique, called *union by rank*, consists in always choosing the root with the greatest rank to be the representative of the union while performing the **Link** operation. If the two canonical elements x and y have the same rank, then one of the elements, say y, is chosen arbitrarily to be the canonical element of the union: y becomes the parent of x; and the rank of y is incremented by one. The rank $\text{Rnk}(x)$ is a measure of the depth of the tree rooted in x, and is exactly the depth of this tree if the path compression technique is not used jointly with the union by rank technique. Union by rank avoids creating degenerate trees, and helps keeping the depth of the trees as small as possible. For a more detailed explanation and complexity analysis, see Tarjan's paper [24].

Procedure `MakeSet(`*element* x`)`

$\text{Par}(x) := x$; $\text{Rnk}(x) := 0$;

Function `element Find(`*element* x`)`

if *(*$\text{Par}(x) \neq x$*)* **then** $\text{Par}(x) := \text{Find}(\text{Par}(x))$;
return $\text{Par}(x)$;

Function `element Link(`*element* x*, element* y`)`

if *(*$\text{Rnk}(x) > \text{Rnk}(y)$*)* **then** exchange($x, y$);
if *(*$\text{Rnk}(x) == \text{Rnk}(y)$*)* **then** $\text{Rnk}(y) := \text{Rnk}(y) + 1$;
$\text{Par}(x) := y$;
return y;

B. Illustration of union-find: labelling the connected components

We can illustrate the use of the union-find algorithm on the classical problem of finding the connected components of a subset X of a graph (V, E). Algorithm 1 (ConnectedComponents) is given below. For a set X, this algorithm returns a map M that gives for each point p, the canonical element $M(p)$ of the connected component of X which contains p.

Algorithm 1: ConnectedComponents

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

During the first pass (loop 1), for each point p of the set X, the set $\{p\}$ is added to the collection $\mathcal{Q}$ of disjoint subsets. Then, loop 2 processes all points of X in an arbitrary order. For each point p, we first find the canonical element of the set it belongs to (line 3). Then, for each neighbor q of p such that $q \in X$ (line 4), we find the canonical element of the set which contains q (line 5). If p and q are not already in the same set, that is if the two canonical elements differ (line 6), then the corresponding sets are merged (line 7), and one of the two canonical elements is chosen to be the canonical element of the merged set. At the end, a simple pass on all the elements of X (loop 8) builds the map M.
Note that, if the vertices can be processed in some very specific order (as the scanline order), the ConnectedComponents algorithm becomes linear [10], [9]. Unfortunately, such a specific strategy is not applicable for the component tree algorithm, where the scanning order depends on the altitudes of the vertices.

C. Component tree algorithm: high-level description

We are now ready to introduce our quasi-linear algorithm for building the component tree $\mathcal{C}(F)$ from a weighted graph $G = (V, E, F)$.

[2]The precise definition of α, a "diagonal inverse" of the Ackermann's function, involves notions which are not in the scope of this paper, it can be found in [24].

The algorithm simulates the emergence process described in the introduction, and maintains several data structures. The main one is a forest, which initially consists of a set of mutually disconnected *nodes*, each node being associated (initially) to a single vertex of the graph G. During the emergence process, which is realized by scanning all the vertices of G by decreasing order of altitude, the vertices which belong to a same component and have the same altitude are grouped together thanks to a disjoint set collection called $\mathcal{Q}_{\text{node}}$. The canonical element of such a set is called a *canonical node*. Notice that the disjoint set collection $\mathcal{Q}_{\text{node}}$ has essentially the same function as the disjoint set collection used by algorithm ConnectedComponents (sec. III-B).

Simultaneously, the canonical nodes are progressively linked together to form *partial trees*, each partial tree represents intuitively an emerged island. At the end of the execution, a unique tree groups all the canonical nodes, each one of these nodes represents a component of G, and the whole tree constitutes the component tree of G. To reach a quasi-linear time complexity, we have to maintain another collection $\mathcal{Q}_{\text{tree}}$ of disjoint sets, and an auxiliary map called *lowestNode*. Given an arbitrary node P, the collection $\mathcal{Q}_{\text{tree}}$ allows to find, in quasi-constant time, a node T which "represents" the partial tree which contains P. Due to the particular management of $\mathcal{Q}_{\text{tree}}$, this node T cannot be guaranteed to be precisely the root of the partial tree, this is why we also need to maintain the map *lowestNode* which associates, to each canonical element of $\mathcal{Q}_{\text{tree}}$, the root of the corresponding partial tree.

D. Component tree algorithm: detailed view

Algorithm 2 (BuildComponentTree) is given below. It uses two auxiliary functions **MakeNode** and **MergeNodes**. To represent a node of $\mathcal{C}(F)$, we use a structure called `node` containing the level of the node, and the list of nodes which are children of the current node. For building the component tree, we do not need the reverse link, that is we do not need to know the parent of a given node, but let us note that such information is useful for applications, and can easily be obtained in a linear-time post-processing step. In what follows, we are going to show how to compute some attributes associated to each node of the component tree; we thus need that the structure `node` contains some fields that store those attributes, namely `level`, `area` and `highest`. We defer both the precise definition of the attributes and the explaination on how they are computed until section VI, in order to concentrate on the component tree itself.

Function `node MakeNode(`*int level*`)`

Allocate a new node n with an empty list of children;
n→ level := level; n→ area := 1; n→ highest := level;
return n;

After a preprocessing (line 1, achievable in linear time for short integers [6]) which sorts the points by decreasing order of level and which prepares the two union-find implementations (line 2), we process the points, starting with the highest ones.

Function `int MergeNodes(`*int node1, int node2*`)`

```
tmpNode := Link_node(node1,node2);
if (tmpNode == node2) then
    Add the list of children of nodes[node1]
        to the list of children of nodes[node2];
    tmpNode2 := node1;
else
    Add the list of children of nodes[node2]
        to the list of children of nodes[node1];
    tmpNode2 := node2;
nodes[tmpNode]→area :=
    nodes[tmpNode]→area + nodes[tmpNode2]→area;
nodes[tmpNode]→highest :=
    max(nodes[tmpNode]→highest,
        nodes[tmpNode2]→highest);
return tmpNode;
```

Let us suppose that we have processed a number of levels. We have built all nodes of the component tree that are above the current level, and we are building the nodes with exactly the current level. For a given point p of the current level (line 3), we know (through the collection $\mathcal{Q}_{\text{tree}}$) the partial tree the node p belongs to (line 4). In each partial tree, there is only one node with the current level, that we can obtain through the auxiliary map $lowestNode$. We then find the associated canonical node (line 5).

We then look at each neighbor q of p with a level greater or equal to the current one (loop 6). Note that, as the graph is symmetric, the "linking operations" between two points are done when one of the two points is processed as a neighbor of the other. Thus, we can use the order of scanning of the points, and we only need to examine the "already processed" neighbors of p. Such a neighbor q satisfies $F(q) \geq F(p)$.

Exactly as we have done for the point p, we search for the canonical node corresponding to the point q (lines 7-8). If the canonical node of p and the canonical node of q differ, that is if the two points are not already in the same node, we have two possible cases:

- either the two canonical nodes have the same level; this means that these two nodes are in fact part of the same component, and we have to merge the two nodes (line 9 and function **MergeNodes**). The merging of nodes of same level is done through the collection $\mathcal{Q}_{\text{node}}$ of disjoint sets. The merging relies on the fact that the **Link$_{\text{node}}$** function always chooses one of the two canonical elements of the sets that are to be merged as the canonical element of the merged set. This fact is used in the sequel of the function.
  Once the merging has been done, one of the nodes is chosen to be the canonical element of the disjoint set. Observe that the other node is not needed anymore. Indeed, we only have to know to which disjoint set this last node belongs to, and the answer to this question is given by the **Find$_{\text{node}}$** function.
- or the canonical node of q is strictly above the current level, and thus this node becomes a child of the current

Algorithm 2: BuildComponentTree

Data: (V, E, F) - vertex-weighted graph with N points.
Result: $nodes$ - array $[0 \dots N-1]$ of nodes.
Result: $Root$ - Root of the component tree
Result: M - map from V to $[0 \dots N-1]$ (component mapping).
Local: $lowestNode$ - map from $[0 \dots N-1]$ to $[0 \dots N-1]$.

```
1  Sort the points in decreasing order of level for F;
2  foreach p ∈ V do {MakeSet_tree(p); MakeSet_node(p); nodes[p]:= MakeNode(F(p)); lowestNode[p] := p;};
3  foreach p ∈ V in decreasing order of level for F do
4      curTree := Find_tree(p);
5      curNode := Find_node(lowestNode[curTree]);
6      foreach already processed neighbor q of p with F(q) ≥ F(p) do
7          adjTree := Find_tree(q);
8          adjNode := Find_node(lowestNode[adjTree]);
           if (curNode ≠ adjNode) then
               if (nodes[curNode]→level == nodes[adjNode]→level) then
9                  curNode := MergeNodes(adjNode, curNode);
               else
                   // We have nodes[curNode]→level < nodes[adjNode]→level
10                 nodes[curNode]→addChild(nodes[adjNode]);
11                 nodes[curNode]→area := nodes[curNode]→area + nodes[adjNode]→area;
12                 nodes[curNode]→highest := max(nodes[curNode]→highest, nodes[adjNode]→highest);
13             curTree := Link_tree(adjTree, curTree);
14             lowestNode[curTree] := curNode;
15 Root := lowestNode[Find_tree(Find_node(0))] ;
16 foreach p ∈ V do M(p) := Find_node(p);
```

node (line 10).

In both cases, we have to link the two partial trees, this is done using the collection $\mathcal{Q}_{\text{tree}}$ (line 13). We also have to keep track of the node of lowest level for the union of the two partial trees, that we store in the array $lowestNode$ (line 14).

At the end of the algorithm, we have to do a post-processing to return the desired result. The root of the component tree can easily be found (line 15) using the array $lowestNode$ and the two disjoint set structures $\mathcal{Q}_{\text{tree}}$ and $\mathcal{Q}_{\text{node}}$. The component mapping M can be obtained using the disjoint set $\mathcal{Q}_{\text{node}}$ (loop 16).

IV. Illustration of the algorithm

Let us illustrate the work of the algorithm on an example. Consider the weighted graph of Fig. 2.a. The points are labelled according to their usual lexicographical order (Fig. 2.b).

At the beginning of the sixth step, we have already constructed parts of the component tree (Fig. 3.b). We show in Fig. 3.a the maps Par_{tree}, Par_{node}, and $lowestNode$. For the maps Par_{tree} and Par_{node}, the canonical elements appear in white. It should be noted that the $lowestNode$ mapping is only used for the canonical elements of Par_{tree}: this explains why the values of $lowestNode$ for other elements (in grey) are not updated.

We are going to process nodes at level 50. The first node at level 50 is node 3. Node 0 is a neighbor of node 3. The canonical node corresponding to 0 is node 1, the level of which

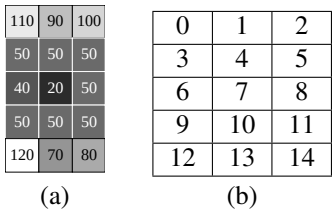


Fig. 2. (a) Original vertex-weighted graph. (b) Points are labelled according to the usual lexicographic order, but they will be processed by decreasing level (that is: $12, 0, 2, 1, 14, 13, 3, 4, 5, 8, 9, 10, 11, 6, 7$).

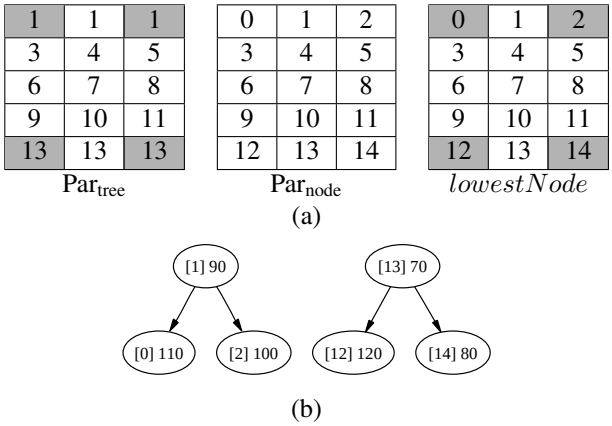


Fig. 3. Beginning of step 6. (a) State of the maps Par_{tree}, Par_{node} and $lowestNode$. (b) Partial trees constructed.

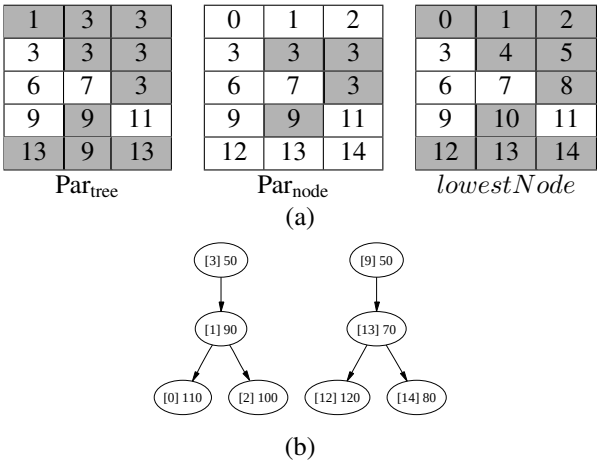


Fig. 4. Beginning of step 11. (a) State of the maps Par_{tree}, Par_{node} and $lowestNode$. (b) Partial trees constructed.

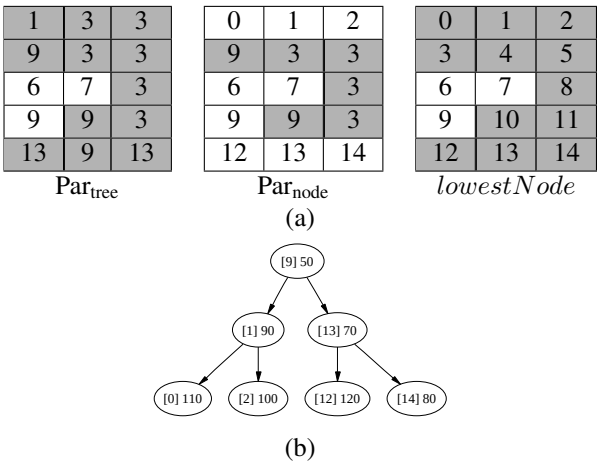


Fig. 5. End of step 11. (a) State of the maps Par_{tree}, Par_{node} and $lowestNode$. (b) Partial trees constructed.

is 90. Thus node 3 becomes the parent of node 1. Then, node 3 is linked for $\mathcal{Q}_{\text{node}}$ succesively with nodes 4, 5 and 8. Then node 9 is examined, and is linked for $\mathcal{Q}_{\text{node}}$ with node 10, the node 9 being chosen as the canonical one. Node 9 is a neighbour of node 12, the canonical element of which is node 13 (level 70). Thus, node 13 becomes a child of node 9. We are then at the beginning of step 11, and this is illustrated on Fig. 4.

Node 11 is a neighbor of both nodes 8 and 10. The canonical node of node 8 is node 3 at level 50. Thus, node 11 and node 3 are linked for $\mathcal{Q}_{\text{node}}$, and node 3 is chosen as the canonical one. The canonical node of node 10 is node 9 at level 50. Thus, nodes 9 and 3 are merged, that is, the corresponding partial trees are merged into a single tree. Node 9 is chosen as the canonical element of the level 50 component, and the children of node 3 are transfered to node 9. We are in the situation depicted in Fig. 5.

We then process node 6 at level 40, which becomes the parent of node 9 at level 50. Node 9 and node 6 are linked for $\mathcal{Q}_{\text{tree}}$, and node 9 is chosen as the canonical element for the partial tree. The lowest node in this partial tree is

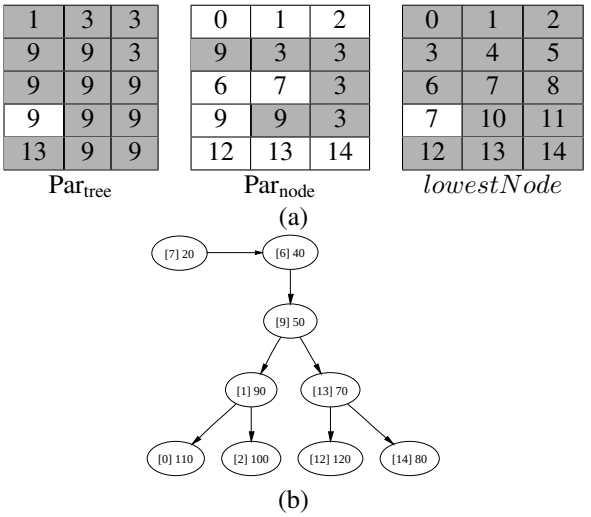


Fig. 6. End of step 14. (a) State of the map Par_{tree}, Par_{node} and $lowestNode$. (b) Component tree.

node 6 at level 40. We use the map $lowestNode$ to store that information, by setting $lowestNode[9] := 6$. Then we process node 7 at level 20, which becomes the parent of node 6. Node 9 is chosen as the canonical element for the partial tree, and thus we have to store the lowest node by setting $lowestNode[9] := 7$. There is no node lower than 20, and thus, the component tree is built. The final situation is depicted in Fig. 6.

The collection $\mathcal{Q}_{\text{tree}}$ of disjoint sets is not useful anymore: indeed, each node of the graph has been examined, and they are all linked for $\mathcal{Q}_{\text{tree}}$, the canonical element being the node 9. The root of the component tree is the node 7. Each of the canonical elements of the collection $\mathcal{Q}_{\text{node}}$ corresponds to a component of F: observe in particular the level 50, whose canonical node is node 9. The collection $\mathcal{Q}_{\text{node}}$ can be used to compute the component mapping M.

V. Complexity analysis

Let n denote the number of points in V, and let m denote the number of edges of the graph (V, E).

The sorting of the points (line 1) can be done in $O(n)$ if the weigths are small integers (counting sort [6]), and in $O(n \log(\log(n)))$ if each weight can be stored in a machine memory word (long integers or floating point numbers [1]).

Loop 2 is the preparation for the union-find algorithm. It is obviously $O(n)$.

In the function **MergeNodes**, the merging of the lists of children can be done in constant time, because we can merge two lists by setting the first member of one list to be the one that follows the last member of the other list. This requires the two lists to be disjoint, which is the case (we are dealing with disjoint sets), and an adequate representation for lists (chained structure with pointers on both first and last element).

The amortized complexity of line 6 is equal to the number m of edges of the graph (V, E). The amortized complexity of all calls to the union-find procedures is quasi-linear (in the sense explained in section III-A) with respect to m. The building of the component mapping M is obviously linear.

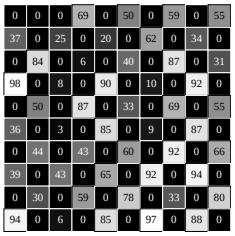


Fig. 7. An example of an artificially generated image of size N*N, where values of pixel (x_1, x_2) with $x_1 + x_2$ odd are uniformly distributed between 0 and $N * N$, and where the other half of the pixels are 0. Using a series of such images, one can verify that the component tree algorithm of Salembier *et al.* is quadratic.

Thus the complexity of the algorithm 2 (BuildComponentTree) is quasi-linear if the sorting step is linear.

Note that the memory for the $lowestNode$ array is not necessary: we can easily modify the code so that we store the content of $lowestNode$ as negative values in Par_{tree} for the canonical element of $\mathcal{Q}_{\text{tree}}$. In this case, for an element $x \in V$, $\text{Find}_{\text{tree}}(x)$ still returns the canonical element c for $\mathcal{Q}_{\text{tree}}$, but $lowestNode(c) = -\text{Par}_{\text{tree}}(c)$. The modifications that have to be made to **MakeSet**, to **Find**, and to **BuildComponentTree** are straightforward and do not change the complexity of the algorithm.

For comparison purpose, one can prove that the most cited component tree algorithm, the Salembier *et al.* algorithm [21] is quadratic. More precisely, although there is no complexity analysis in [21], one can verify that the Salembier *et al.* algorithm has a worst-case time complexity in $O(n \times h + m)$ where h is the number of levels of the image. The worst case can be attained using a series of artificially generated images such that half of the pixels are maxima of the images (an example of an image of the series is provided in Fig. 7). However, this worst case is rare in practice. We observe that, when the level of a point is a short integer (between 0 and 255), the Salembier *et al.* algorithm is generally twice as fast as our algorithm. This can be explained by the fact that, for each point of the image, we have to access the two union-find data structures, while this is not the case for the Salembier *et al.* algorithm.

VI. Attributes

A major use of the component tree is for image filtering: for example, we may want to remove from an image the "lobes" that are not "important enough" or "negligible". Such an operation is easy to do by simply removing the "negligible" components of the component tree. To make such an idea practicable, it is necessary to quantify the relative importance of each node of the component tree. We can do that by computing some attributes for each node.

Among the numerous attributes that can be computed, three are natural: the height, the area, and the volume (Fig. 8).

Let $[k, c] \in \mathcal{C}(F)$. We define

$$\begin{aligned} \text{height}([k,c]) &= \max\{F(x) - k + 1 | x \in c\} \\ \text{area}([k,c]) &= \text{card}(c) \end{aligned}$$

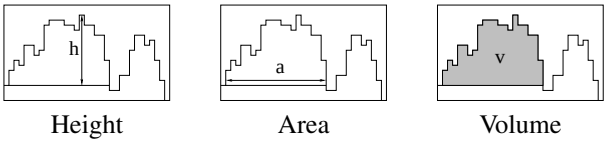


Fig. 8. Illustration of the height, the area and the volume of a component.

$$\text{volume}([k,c]) = \sum_{x \in c} (F(x) - k + 1)$$

The area is easy to compute while building the component tree. Each time two components merge (*i.e.* in the function MergeNodes) or each time a component is declared the parent of another one (*i.e.* line 11 of algorithm 2 BuildComponentTree), we keep as the new area the sum of the areas of the two components.

For computing the highest level in the component, we do as we did for the area, replacing the sum by the maximum (see line 12 of algorithmm 2 BuildComponentTree and the function MergeNodes). From this highest level, the height of a component n can easily be computed by setting $\text{height}(n) = (n \rightarrow \text{highest}) - (n \rightarrow \text{level}) + 1$.

To compute the volume, we first need the area. We then apply the recursive function ComputeVolume on the root of the tree. The complexity of this function is linear with respect to the number of nodes.

Function `int ComputeVolume(`*int n*`)`

```
vol := nodes[n]→area;
foreach c child of nodes[n] do
    vol := vol + ComputeVolume(c) +
        c→area * (c→level - nodes[n]→level);
nodes[n]→volume := vol ;
return vol;
```

VII. Example of application and conclusion

We have mentioned a simple use of the component tree for filtration (removing nodes of the tree whose attribute is below a given threshold). A more advanced use consists in finding the most significant lobes of a given weighted graph F. More precisely, we want to find the N most significant components with respect to either the height, area or volume criterion. By using the tree, this task reduces to the search of the N nodes that have the largest attribute values and are not bound with each other (even transitively) by the inclusion relation. Algorithm 3 (Keep_N_Lobes) performs this task. Its time complexity is in $O(\text{sort}(n) + m)$, where m is the number of vertices in the graph, n is the number of component tree nodes and $\text{sort}(n)$ is the complexity of the sorting algorithm. At the end of the algorithm, the remaining leaves (more precisely, the pixels which are associated to these leaves) mark the desired significant lobes. For this algorithm, each node must include fields to store its parent and its number of children (but the list of children of a given node is not necessary).

Fig. 9 illustrates this algorithm. Fig. 9.a is an image of cell, in which we want to extract the ten bright lobes. Fig. 9.b shows

Algorithm 3: Keep_N_Lobes

Data: A vertex-weighted graph (V, E, F), its component tree T with attribute value for each node, and the associated component mapping M
Data: The number N of wanted lobes.
Result: The filtered map F

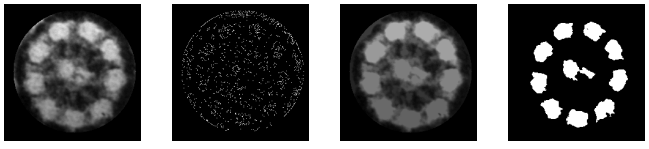
```
Sort the nodes of T by increasing order of attribute value;
Q := ∅; L := number of leaves in T;
forall n do nodes[n]→mark := 0;
while L > N do
    Choose a (leaf) node c in T with smallest attribute value;
    p := nodes[c]→parent ;
    nodes[p]→nbChildren := nodes[p]→nbChildren-1;
    if (nodes[p]→nbChildren > 0) then L := L-1;
    nodes[c]→mark := 1 ; Q := Q ∪ {c};
while ∃c ∈ Q do
    Q := Q \ {c}; RemoveLobe(c);
foreach x ∈ V do F(x) := nodes[M[x]]→level;
```

Function `int RemoveLobe(int n)`

```
if (nodes[n]→mark == 1) then
    nodes[n] := nodes[RemoveLobe(nodes[n]→parent)];
return n;
```

that the image 9.a contains numerous maxima. Fig. 9.c is the filtered image obtained by using algorithm 3 with the volume attribute and with parameter value 10, and Fig. 9.d shows the maxima of this filtered image. Note that a similar result could be obtained with this image by performing attribute based operations using several volume threshold values, following *e.g.* a dichotomic method, until the desired number of maxima is reached. This latter approach is not only less efficient than the proposed algorithm, but it may also fail to find the precise number of maxima required by the user, in the case of components having precisely the same attribute value. In such cases, the proposed algorithm always makes a choice in order to fulfill the user's requirement.

The component tree allows the efficient implementation of complex image and signal filtering, based for example on the use of criteria such as area, volume or depth, or even the use of non-increasing criteria [21]. Although some of these filters may be computed using specific and sometimes faster algorithm (in particular area filtering [17]), using the component tree is in general the simplest and the most efficient way to compute these filters. Moreover, once the component tree of a function is computed, any of these filters, with any parameter value, can be computed at a very low cost. The component tree is also a key element of an efficient algorithm for the topological watershed [8]. New classes of filters, such as second-order connected operators [23] have been recently introduced to generalize connected operators [22]. Those operators can also be efficiently implemented using the component tree [20]. In this paper, we have proposed a simple-to-implement quasi-linear algorithm for computing the component tree. We hope that such an algorithm will facilitate the extensive practical use of such operators.



(a) (b) (c) (d)

Fig. 9. (a) Original image. (b) Maxima of image (a), in white. (c) Filtered image. (d) Maxima of image (c), which correspond to the ten most significant lobes of the image (a).

REFERENCES

[1] A. Andersson, T. Hagerup, S. Nilsson, and R. Raman. Sorting in linear time? In *STOC: ACM Symposium on Theory of Computing*, 1995.
[2] G. Bertand. On the dynamics. *Image and Vision Computing*. Submitted.
[3] G. Bertrand. On topological watersheds. *JMIV*, 22(2-3):217–230, 2005.
[4] E.J. Breen and R. Jones. Attribute openings, thinnings and granulometries. *Comp. Vision and Image Und.*, 64(3):377–389, Nov. 1996.
[5] Y.-J. Chiang, T. Lenz, X. Lu, and G. Rote. Simple and optimal output-sensitive construction of contour trees using monotone paths. *Comp. Geometry: Theory and Applications*, 30(2):165–195, 2005.
[6] T. H. Cormen, C. L. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 1990.
[7] M. Couprie and G. Bertrand. Topological grayscale watershed transform. In *SPIE Vision Geom. V Proc.*, volume 3168, pages 136–146, 1997.
[8] M. Couprie, L. Najman, and G. Bertrand. Quasi-linear algorithms for the topological watershed. *JMIV*, 22(2-3):231–249, 2005.
[9] M.B. Dillencourt, H. Samet and M. Tamminen. *A general approach to connected-component labeling for arbitrary image representations*. Journal of the Assoc. Comput. Mach., 39(2):253–280, 1992.
[10] Ch. Fiorio and J. Gustedt. *Two linear Union-Find strategies for image processing*. *Th. Computer Science* 154:165-181, 1996.
[11] P. Guillataud. *Contribution à l'analyse dendroniques des images*. PhD thesis, Université de Bordeaux I, 1992.
[12] P. Hanusse and P. Guillataud. Sémantique des images par analyse dendronique. In *8th RFIA*, volume 2, pages 577–588, 1992
[13] J.A. Hartigan. Statistical theory in clustering. *Journal of Classification*, 2:63–76, 1985.
[14] R. Jones. Component trees for image filtering and segmentation. In *NSIP'97*, 1997.
[15] J. Mattes and J. Demongeot. Efficient algorithms to implement the confinement tree. In *LNCS*:1953, pages 392–405. Springer, 2000.
[16] J. Mattes, M. Richard, and J. Demongeot. Tree representation for image matching and object recognition. In *LNCS*:1568, pages 298–309, 1999.
[17] A. Meijster and M.H.F. Wilkinson. A comparison of algorithms for connected set openings and closings. *IEEE Trans. on PAMI*, 24(4):484–494, April 2002.
[18] P. Monasse. *Morphological representation of digital images and application to registration*. PhD thesis, Paris-Dauphine Univ., June 2000.
[19] L. Najman and M. Couprie. Quasi-linear algorithm for the component tree. In *SPIE Vision Geometry XII*, Vol. 5300 pages 98–107, 2004
[20] G.K. Ouzounis and M.H.F. Wilkinson. Second-order connected attribute filters using max-trees. In *Mathematical morphology: 40 years on*, pages 65–74. Springer, 2005.
[21] P. Salembier, A. Oliveras, and L. Garrido. Anti-extensive connected operators for image and sequence processing. *IEEE Trans. on Image Proc.*, 7(4):555–570, April 1998.
[22] P. Salembier and J. Serra. Flat zones filtering, connected operators and filter by reconstruction. *IEEE Tr. on Im. Proc.*, 3(8):1153–1160, 1995.
[23] J. Serra. Connectivity on complete lattices. *JMIV*, 9:231–251, 1998.
[24] R.E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22:215–225, 1975.
[25] C. Vachier. *Extraction de caractéristiques, segmentation d'images et Morphologie Mathématique*. PhD thesis, ENSMP, 1995
[26] D. Wishart. Mode analysis: A generalization of the nearest neighbor which reduces chaining effects. In A.J. Cole, editor, *Numerical Taxonomy*, pages 282–319, London, 1969. Academic Press.