RUMINATIONS ON TARJAN'S UNION-FIND ALGORITHM AND CONNECTED OPERATORS

Thierry Géraud

EPITA Research and Development Laboratory (LRDE)
14-16 rue Voltaire, F-94276 Le Kremlin-Bicêtre, France
Phone: +33 1 53 14 59 47, Fax: +33 1 53 14 59 22
thierry.geraud@lrde.epita.fr

Abstract This papers presents a comprehensive and general form of the Tarjan's union-find algorithm dedicated to connected operators. An interesting feature of this form is to introduce the notion of separated domains. The properties of this form and its flexibility are discussed and highlighted with examples. In particular, we give clues to handle correctly the constraint of domain-disjointness preservation and, as a consequence, we show how we can rely on "union-find" to obtain algorithms for self-dual filters approaches and levelings with a marker function.

Keywords: Union-find algorithm, reconstructions, algebraic openings and closings, domain-disjointness preservation, self-dual filters, levelings.

Introduction

Connected operators have the important property of simplifying images while preserving contours. Several sub-classes of these operators have been formalized having stronger properties [8] and numerous applications have been derived from them, e.g., scale-space creation and feature analysis [17], video compression [14], or segmentation [10]. The behavior of connected operators is to merge most of the flat zones of an input image, thus delivering a partition which is much coarser than the input one. In that context, a relevant approach to implement such operators is to compute from an input image the resulting partition. The Tarjan's Union-Find Algorithm, *union-find* for short, computes a forest of disjoint sets while representing a set by a tree [16]. A connected component of points or a flat zone is thus encoded into a tree; a point becomes a node and a partition is a forest. union-find has been used to implement some connected operators; among them, connected component labeling [2], a watershed transform [6], algebraic closing and opening [19], and component tree

computation [4, 11]. A tremendous advantage of union-find lies in its *simplicity*. However, the descriptions of morphological operators relying on this algorithm are usually spoiled by the presence of too many implementation details.

This paper intends to provide the image processing community with a simple and general form of union-find, which is highly adaptable to the large class of connected operators. We show that the description of a given operator with union-find is actually straightforward, comprehensive, and takes very few code. We also present how union-find can be used for the connected operators θ which verify a domain disjointness preservation property. Consequently we show that union-find is a simple way to get algorithms for folding induced self-dual filters [5], the inf-semilattice approach to self-dual morphology [3], and levelings defined on two functions [10].

In order to keep implementation details away from algorithmic considerations, we do not address any single optimization issue. Moreover, we do not enter into a comparison between union-find-based algorithms and other approaches; for those subjects, the reader can refer to [13, 7]. We claimed in [1] that our generic C++ image processing library, Olena [12], has been designed so that algorithms can easily be translated into programs while remaining very readable. To sustain this claim, programs given in this paper rely on our library and, thanks to it, they efficiently run on various image structures (signals, 2D and 3D images, graphs) whatever their data types (Boolean, different integer encodings, floating values, etc.)

In the present document we start from the simplest operator expressed with union-find, namely a connected component labeling, in order to bring to the fore the properties of union-find-based algorithms (Section 1). We stress on the notions of domains and of disjointness-preservation and we present a general formulation of union-find (Section 2). In the second part of this document we give a commented catalogue of connected operators with the help of that formulation (Section 3). Last we conclude (Section 4).

1. Practicing on Connected Component Labeling

In union-find, a connected component is described as a tree. At any time of the algorithm computation, each existing component has a canonical element, a root point at the top of the tree. A link between a couple of nodes within a tree is expressed by a parent relationship. A convenient way to handle the notion of "parent" is to consider that parent is an image whose pixel values are points—given a point x, parent[x] is a point—and that a root point is its own parent. Finding the root point of a component recursively goes, starting from a point of this component, from parent to parent until reaching the root point.

Let us practice first on connected component labeling. The union-find algorithm is composed of an initialization followed by two passes. The first one aims at computing the tree forest and the second one aims at labeling.

```
void init() {
  is_processed = false; // that is, for all points
  cur_label = 0;
  O = false; // background is the default value
}

void first_pass() {
  bkd_scan p(D); // R_D is the bkd scan
  nbh_scan n(nbh);
  for_all (p)
    if ( is_in_I(p) ) { // "body 1.1"
      make_set(p); // so {p} is a set
      for_all_neighbors ( n, p )
        if ( D.holds(n) ) // n belongs to D
          if ( is_processed[n] )
            do_union(n, p);
      is_processed[p] = true;
    }
}

void second_pass() {
  fwd_scan p(I); // versus bkd in 1st pass
  for_all (p)
    if ( is_in_I(p) ) { // "body 2.1"
      if ( is_root(p) )
        set_O_value(p); // new label
      else
        O[p] = O[parent[p]]; // propagation
    }
}
```

In the first pass, points of the input set I are browsed in a given order. In the case of image processing, the domain D of the input image includes I. Practically, I is represented by its membership function defined over D and is encoded as a binary image. A convenient way to browse elements of I is thus performed by a classical forward or backward scan of the points p of D and testing if p is in I is required. Let us denote by $\mathcal{R}_D$ the ordering of points of D which corresponds to the way we browse D during the first pass. The current point p is first turned into a set: $\{p\}$. Neighbors of p are then inspected

to eventually merge p with an existing tree. Let us denote by n a neighbor of p such as $n \in I$ (actually we have to ensure that n actually lies in the image domain D to prevent problems when p happens to belong to the internal boundary of D). If n has not been processed yet, it does not make sense to inspect n since it does not belong to a tree structure. In the other case, we proceed to the union of $\{p\}$ and of the set to which n belongs.

A key point of union-find lies in this task; performing the union of a couple of trees is a single simple operation: linking their respective root points, say r_1 and r_2 with $r_1 \mathcal{R}_D r_2$, so that the parent of r_1 is r_2. This rule ensures a very strong property: when processing p in the first loop, $\forall p'$ such as $p' \mathcal{R}_D p$, we have $p' \mathcal{R}_D \, parent(p')$ and $parent(p') \mathcal{R}_D \, p$. Adding the point p to the connected component that contains n is then as simple as setting to p the parent of r, where r is the root point of $\Gamma_I(n)$.

The second pass of union-find browses points of D in the reverse order, $\mathcal{R}_D^{-1}$, as compared to the first pass. Thanks to parenthood construction, the following two properties hold. 1) For any component (or flat zone) computed during the first pass, the first point of this component visited during the second pass is the root point. 2) In the second pass, a point is always visited after its parent. So, for each point $p \in I$ using $\mathcal{R}_D^{-1}$, we assign p a new label in the output image O if p is root, otherwise we assign p the same label as its parent.

```
bool is_in_I(point p) { return I[p] == true; }

void make_set(point p) { parent[p] = p; }

bool is_root(point p) { return parent[p] == p; }

point find_root(point x) {
  if ( is_root(x) ) return x;
  else return ( parent[x] = find_root(parent[x]) );
}

void set_parent(point r, point p) { parent[r] = p; }

void do_union(point n, point p) {
  point r = find_root(n);
  if ( r != p )
    set_parent(r, p);
}

L set_O_value(point p) { return ++cur_label; }
```

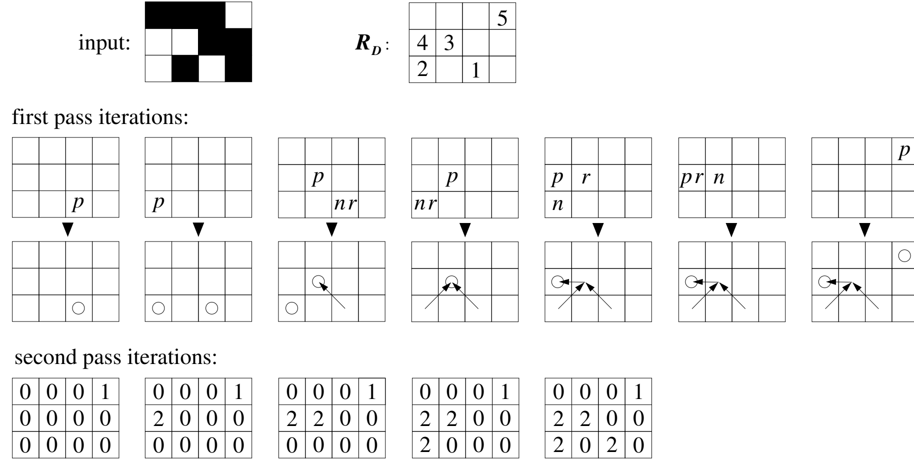

Figure 1. Connected component (8-connectivity) labeling using union-find: arrows represent parenthood and circles depict root points.

Both passes of connected component labeling using union-find are illustrated in Figure 1.

2. Introducing Domains and Disjointness-Preservation

The main characteristic of union-find appears to be in its overall structure. First, let us take a partition of the image domain D into $m+1$ disjoint sets defined by: $D = D' \cup D''$ with $D' = \left(\cup_{i=1}^{m} D_i\right)$. In this partition D'' is the set of points of D that are not subject to forest computation.

A requirement about D'' is therefore that, $\forall p \in D''$, the value $O[p]$ can be directly computed from the operator input. So we compute O over D'' as a whole in the initialization part of union-find (set_default_O routine). Also note that we will never have proper values for parenthood in D''.

A sub-domain D_i with $i = 1..m$ can have its own definition of forest computation, different from the ones of D_j with $j \neq i$. Consider for instance the simultaneous connected component labeling of both object and background in a binary 2D image; obviously we have $D'' = \emptyset$, $D_1 = I$, and $D_2 = I^c$. Processing two forests then allows us to rely on two distinct neighborhoods so that topological inconsistency is avoided. Keeping this idea in mind, the description of union-find mutes into a more general form depicted in Figure 2. Both first and second passes process in an independent way the domains D_i thanks to a test is_in_Di; that gives rise to the "*body 1.i*" and "*body 2.i*" sections. Furthermore, some variations between bodies in a same pass are now conceivable.

```
void init() {
  is_processed = false;
  set_default_O();
}

void first_pass() {
  // declarations are removed
  for_all (p)
    if ( is_in_D1(p) ) // body 1.1
    else if ( is_in_D2(p) ) // body 1.2
    // other bodies 1.i ...
}

void second_pass() {
  // declarations are removed
  for_all (p)
    if ( is_in_D1(p) ) // body 2.1
    else if ( is_in_D2(p) ) // body 2.2
    // other bodies 2.i ...
}

// with body 1.i being:
{
  make_set(p);
  for_all_neighbors(n, p)
    if ( D.holds(n) )
      if ( is_in_Di(n) and is_processed[n] )
        do_union(n, p);
      // optional:
      else visit_extB_of_Di(n, p);
  set_processed(p);
}

// with body 2.i being:
{
  if ( is_root(p) ) set_O_value_in_Di(p);
  else O[p] = O[parent[p]];
}
```

Figure 2. General form of union-find with domains.

Preserving domain disjointness. A strong assumption is implicitly managed in the writing of "*body 1.i*" due to the tests "is_in_Di(n)": we are preserving disjointness over domains, that is, each connected component (or flat zone) Γ created by such an algorithm cannot cross domains frontiers. We have:

$$\forall \Gamma \in O,\ \exists i \ \text{ such as } \ \Gamma \subseteq D_i \ \text{ and } \ \forall j \neq i,\ \Gamma \cap D_j = \emptyset.$$

Visiting domain boundaries. During the first pass, while processing a neighbor n of p, if we do not enter component computation, that means that n does not belong to D_i. If $m \geq 2$, that also means that n can have already been processed or not. Since $p \in D_i$, n belongs to the external boundary $\mathcal{B}^{ext}(D_i)$ of D_i and, when the first pass is completed, we have visited all the points of $\mathcal{B}^{ext}(D_i)$. This general version of union-find thus get us the ability of fetching information from $\mathcal{B}^{ext}(D_i)$ (through the visit_extB_of_Di routine).

Attaching auxiliary data to components. A connected component encoded as a tree is represented by its root point. If one intends to implement with union-find a particular operator which requires information on components, some auxiliary data just have to be attached to every root points. Furthermore, we have the ability to attach to components a distinct type of data per domain. So we have to adapt the routines which deal with parenthood in the first pass:

```
void make_set_in_Di(point p) {
   parent[p] = p; // creation of {p}
   init_data_in_Di(p); // so p has data
}
void set_parent_in_Di(point r, point p) {
   parent[r] = p; // 2 components are now connected
   merge_data_in_Di(r, p); // so p carries data
}
```

Last, please remark that updating data is possible while visiting boundaries $\mathcal{B}^{ext}(D_i)$ (first pass) and that data are usually expected to influence the operator output (second pass, routine set_O_value_in_Di, called when p is root).

Extension to morphology on functions. In the field of morphology on grey-level functions, given a function f, two trivial orderings between points of D can be derived from $\mathcal{R}_D$ and from the one of the complete lattice framework: $p\,\mathcal{R}^{\uparrow} p' \;\Leftrightarrow\; f(p) < f(p')$ or $(\, f(p) = f(p')$ and $p\,\mathcal{R}_D\, p'\,)$ and its reverse ordering $\mathcal{R}^{\downarrow}$. If we choose $\mathcal{R}^{\uparrow}$ for the first pass of union-find, the evolution of connected components during this pass mimics a flooding of f and we get an extensive behavior of the connected operator. By duality, we obtain an anti-extensive behavior with $\mathcal{R}^{\downarrow}$. In the literature about implementing connected operators with union-find, namely algebraic openings and closings

in [19, 7], the notion of domains is absent and the whole image domain is processed. We actually have $D = D'$ (so $D'' = \emptyset$) and at first glance that seems relevant. A novelty here appears during the first pass: connected components can grow even by merging flat zones of the input image. The expansion of a component is stopped when a given increasing criterion is no more satisfied; this component then turns "inactive". That leads us to:

```
void init_data_in_Di(point p) {
  is_active[p] = true; // and handle other data...
}
void do_union_in_Di(point n, point p) {
  point r = find_root(n);
  if ( r != p )
    if ( is_flat_in_Di(r, p) or equiv_in_Di(r, p) )
      set_parent_in_Di(r, p);
}
bool equiv_in_Di(point r, point p) {
  if ( not is_active[r] or not is_active[p] )
    return false;
  if ( not satisfies_criterion_in_Di(r, p) ) {
     is_active[p] = false;
     return false;
  }
  return true;
}
```

Connected operators relying on two functions. So far, just notice that, changing the overall structure of union-find from the first way of browsing points of D to the second one (see below) does not affect the result of the algorithm thanks to the domain disjointness preservation property.

```
// first way of browsing points
  for_all (p)
         if ( is_in_D1(p) ) // body *.1
    else if ( is_in_D2(p) ) // body *.2
    // ...

// second way of browsing points
  for_all (p_in_D1) // body *.1
  for_all (p_in_D2) // body *.2
  // ...
```

3. A Catalogue of Union-Find-Based Connected Operators

In the previous section, we have presented a general form for union-find that relies on a separation between domains. We then just have to fill the holes of this form to get algorithms that implement some connected operators.

Morphology on Sets with Union-Find

Let us first take as an example the reconstruction by dilation operator, R^δ, which applies on a two sets, a marker F and a mask G such as $F \subseteq G$. In the following, given a set X and a point $x \in X$, we will denote by Γ_X a connected component of X. Denoting $O = R^\delta_G(F)$, we have the property $O \subset G$ and, since R^δ is a connected operator with respect to the mask, $\{\Gamma_O\} \subseteq \{\Gamma_G\}$. A first obvious idea for using union-find is then to compute the connected components of G and search for those that belong to O. So we have $D' = G$ and $\Gamma_{D'}$ are computed. However, we can consider the operator input as a four-valuated function $I_{F,G}$ defined on D, such as the value $I_{F,G}(p)$ indicates whether $p \in D$ is included in $F \cap G$, $F \cap G^c$, $F^c \cap G$, or $F^c \cap G^c$. That leads to different ways of using union-find to implement reconstructions depending on the choice of a partition of D into $D' \cup D''$.

DEFAULT	INIT(p)	BORDER(n,p)	MERGE(r,p)
R^δ obvious version: $D' = G \rightsquigarrow (\Gamma_{D'} \subseteq O \Leftrightarrow \exists p \in \Gamma_{D'},\, p \in F)$			
O = **false**;	O[p] = F[p];		O[p] = O[r] **or** O[p];
R^δ alternative version: $D' = F^c \cap G \rightsquigarrow (\Gamma_{D'} \subseteq O \Leftrightarrow \exists n \in \mathcal{B}^{ext}(\Gamma_{D'}),\, n \in F)$			
O = F;	O[p] = **false**;	**if** (F[n]) O[p] = **true**;	O[p] = O[r] **or** O[p];
R^ϵ dual version of "R^δ with $D' = G$": $D' = G^c \rightsquigarrow (\Gamma_{D'} \subseteq O \Leftrightarrow \forall p \in \Gamma_{D'},\, p \in F)$			
O = **true**;	O[p] = F[p];		O[p] = O[r] **and** O[p];
R^ϵ alternative version: $D' = F \cap G^c \rightsquigarrow (\Gamma_{D'} \subseteq O \Leftrightarrow \forall n \in \mathcal{B}^{ext}(\Gamma_{D'}),\, n \in F)$			
O = G;	O[p] = **true**;	**if** (**not** F[n]) O[p] = **false**;	O[p] = O[r] **and** O[p];

The table above presents for several operators the respective definitions of the following routines (from left to right): set_O_default_in_Di, init_data_in_Di, visit_extB_of_Di, and merge_data_in_Di. Last, when the only auxiliary data required to implement an operator represent the Boolean evaluation of $[\Gamma_{D'} \subseteq O]$, the output image can store these data as an attachment to root points and set_O_value has nothing left to perform. From our experiments, an appropriate choice for D'—depending on *a priori* knowledge about F and G—makes the union-find-based approach a serious competitor of the efficient hybrid algorithm proposed in [18]. Last, the case of regional extrema identification is summarized in the table below.

DEFAULT	INIT(p)	BORDER(n,p)	MERGE(r,p)
Regional minima identification of a function f with $D' = D$ (with either $\mathcal{R}^\downarrow$ or $\mathcal{R}^\uparrow$)			
	O[p] = **true**;	**if** (f[n] < f(p)) O[p] = **false**;	O[p] = O[r] **and** O[p];
Regional minima identification with $D' = D$ which relies on $\mathcal{R}^\downarrow$			
	O[p] = **true**;	**if** (is_processed[n] **and** f[n] != f(p)) O[p] = **false**;	O[p] = O[r] **and** O[p];

Extension to Functions

For some connected operators on functions that deliver functions, the table below recaps their corresponding definitions with union-find; the columns CRIT and VALUE respectively depict the result returned by satisfies_criterion and the body of set_O_value. For reconstructions, f and g being the input marker and mask functions, we compute flat zones from g, which is flattened by the operator, is_flat(r,p) returns g[r] == g[p].

INIT(p)	MERGE(r,p)	CRIT(r,p)	VALUE(p)
R^δ (f marker and g mask such as $f \leq g$); $\mathcal{R}^\downarrow$ is mandatory since g is lowered			
o[p] = f[p];	o[p] = max(o[r], o[p]);	g[p] >= o[r]	**if** (**not** is_active[p]) o[p] = g[p];
R^ϵ (f marker and g mask such as $f \geq g$); $\mathcal{R}^\uparrow$ is mandatory since g is "upper-ed"			
o[p] = f[p];	o[p] = min(o[r], o[p]);	g[p] <= o[r]	**if** (**not** is_active[p]) o[p] = g[p];
Area opening (resp. closing) of a function f; $\mathcal{R}^\downarrow$ (resp. $\mathcal{R}^\uparrow$) is mandatory			
area[p] = 1;	area[p] = area[r] + area[p];	area[r] < λ	o[p] = f[p];
Volume opening (resp. closing) of f; $\mathcal{R}^\downarrow$ (resp. $\mathcal{R}^\uparrow$) is mandatory			
area[p] = 1; vol[p] = 1;	vol[p] = vol[r] + vol[p] + (area[p] $*$ abs(f[r]$-$f[p])) area[p] = area[r] + area[p];	vol[r] < λ	o[p] = f[p];

Operators Relying on Two Functions

Many morphological operators over two functions, f and g, have been defined from a couple of connected operators, φ extensive and ψ anti-extensive, following the general formulation [8, 5, 3]:

$$[\theta(f,g)](p) = \begin{cases} [\varphi(f,g)](p) & \text{if } p \in D^\uparrow(f,g) \\ [\psi(f,g)](p) & \text{if } p \in D^\downarrow(f,g) \\ f(p) & \text{otherwise.} \end{cases}$$

under the constraint of being disjointness-preservative regarding $D^\uparrow(f,g)$ and $D^\downarrow(f,g)$, the domains of D where θ is *expected* to be respectively extensive and anti-extensive. Let us denote by $D^\circ(f,g) = (D^\uparrow(f,g) \cup D^\downarrow(f,g))^c$ the domain where θ is expected to be constant. In the following we will elude in domain names the dependence upon f and g since that does not lead to any ambiguity. By extension, let us introduce $D^{\overset{\circ}{\downarrow}} = D^\downarrow \cup D^\circ$ and $D^{\underset{\circ}{\uparrow}} = D^\uparrow \cup D^\circ$.

Relying on union-find to get an algorithm starts with choosing domains such as $D = D' \cup D'' = (\cup_i D_i) \cup D''$ and $\forall j \neq i,\ D_j \cap D_i = \emptyset$. As a constraint we

do not want to use the ability of union-find to visit domain boundaries. Since θ is *not* disjointness-preservative with respect to $D^{\uparrow}$ and D°, and to $D^{\downarrow}$ and D°, we cannot obtain a correct result with union-find when we consider setting the domains D_i with any combination of $D^{\uparrow}$, D°, and $D^{\downarrow}$. So far we are in a dead end.

Let us imagine that we relax the disjointness constraint of union-find (!) to form $D_1 = D^{\uparrow\circ}$ and $D_2 = D^{\downarrow\circ}$. The only weird aspect of this idea is that points of D° have to be processed twice during the first pass of union-find. For that, we just have to add a "refresh" step for the points of D° just after handling $D^{\uparrow\circ}$ during the first pass, so that $D^{\downarrow\circ}$ can be properly processed. This single modification is handled as follows:

```
for_all (p_in_D_upper_or_equal)
  // body 1.1
for_all (p_in_Do)
  is_processed[p] = false; // so p can be handled again
for_all (p_in_D_lower_or_equal)
  // body 1.2
```

and the second pass as described in Figure 2 remains unchanged. Although we have introduced a bond between domains (we have $D_1 \cap D_2 \neq \emptyset$), we do not have introduced any inconsistency in parenthood. Put differently, this modified version of union-find does not compute irrelevant components or flat zones.

We can now reuse the descriptions given previously of union-find-based operators to build levelings with markers as defined in [9], a domain-preserving self-dual reconstruction, and partial self-dual operators defined with the inf-semilattice approach in [3]. Results are summarized in the table below.

$D^{\uparrow\circ}$			$D^{\downarrow\circ}$		
$\mathcal{R}_D$	sub-domain	sub-operator	$\mathcal{R}_D$	sub-domain	sub-operator
some levelings g' of g given a function f such as $g' \in Inter(g, f)$ (see [10])					
$\mathcal{R}^{\downarrow}$ on g	$f(p) \leq g(p)$	γ, any lower leveling	$\mathcal{R}^{\uparrow}$ on g	$f(p) \geq g(p)$	ϕ, any upper leveling
domain-preserving self-dual reconstruction with f marker and g mask					
$\mathcal{R}^{\downarrow}$ on g	$f(p) \leq g(p)$	$R^{\delta}_{g}(f)$	$\mathcal{R}^{\uparrow}$ on g	$f(p) \geq g(p)$	$R^{\epsilon}_{g}(f)$
inf-semilattice approach (input function is f) with any anti-extensive operator ψ					
$\mathcal{R}^{\downarrow}$ on $-f$	$f(p) \leq 0$	$-\psi(-f)$	$\mathcal{R}^{\downarrow}$ on f	$f(p) \geq 0$	$\psi(f)$

4. Conclusion

We have presented a general formulation of Tarjan's union-find algorithm so that many connected operators can be straightforwardly mapped into algorithms; we definitely believe that this particular formulation can ease to express new segmentation methods using connected operators relying on union-find.

References

[1] J. Darbon, T. Géraud, and A. Duret-Lutz. Generic implementation of morphological image operators. In *Mathematical Morphology, Proc. of ISMM*, pages 175–184. Sciro, 2002.

[2] M. Dillencourt, H. Samet, and M. Tamminen. A general approach to connected-components labeling for arbitrary image representations. *Journal of the ACM*, 39(2):253–280, 1992.

[3] H. Heijmans and R. Keshet. Inf-semilattice approach to self-dual morphology. *Journal of Mathematical Imaging and Vision*, 17(1):55–80, 2002.

[4] W. H. Hesselink. Salembier's min-tree algorithm turned into breadth first search. *Information Processing Letters*, 88(1–2):225–229, 2003.

[5] A. Mehnert and P. Jackway. Folding induced self-dual filters. In *Mathematical Morphology and its Applications to Image and Signal Processing*, pages 99–108, 2000.

[6] A. Meijster and J. Roerdink. A disjoint set algorithm for the watershed transform. In *EUSIPCO IX European Signal Processing Conference*, pages 1665–1668, 1998.

[7] A. Meijster and M. Wilkinson. A comparison of algorithms for connected set openings and closings. *IEEE Trans. on PAMI*, 24(4):484–494, 2002.

[8] F. Meyer. From connected operators to levelings. In *Mathematical Morphology and its Applications to Image and Signal Processing*, pages 191–198. Kluwer, 1998.

[9] F. Meyer. The levelings. In *Mathematical Morphology and its Applications to Image and Signal Processing*, pages 199–206. Kluwer, 1998.

[10] F. Meyer. Levelings, image simplification filters for segmentation. *Journal of Mathematical Imaging and Vision*, 20(1–2):59–72, 2004.

[11] L. Najman and M. Couprie. Quasi-linear algorithm for the component tree. In *IS&T/SPIE Symposium on Electronic Imaging, In Vision Geometry XII*, pages 18–22, 2004.

[12] Olena. Generic C++ image processing library, `http://olena.lrde.epita.fr`, free software available under GNU Public Licence, EPITA Research and Development Laboratory, France, 2005.

[13] J. B. Roerdink and A. Meijster. The watershed transform: Definitions, algorithms and parallelization strategies. *Fundamenta Informaticae*, 41(1-2):187–228, 2000.

[14] P. Salembier and J. Ruiz. On filters by reconstruction for size and motion simplification. In *Mathematical Morphology, Proc. of ISMM*, pages 425–434. Sciro Publishing, 2002.

[15] P. Soille. *Morphological Image Analysis*. Springer-Verlag, 1999.

[16] R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22(2):215–225, 1975.

[17] C. Vachier. Morphological Scale-Space Analysis and Feature Extraction. In *IEEE Intl. Conf. on Image Processing*, volume 3, pages 676–679, October 2001.

[18] L. Vincent. Morphological grayscale reconstruction in image analysis: Applications and efficient algorithms. *IEEE Trans. on Image Processing*, 2(2):176–201, 1993.

[19] M. Wilkinson and J. Roerdink. Fast morphological attribute operations using tarjan's union-find algorithm. In *Mathematical Morphology and its Applications to Image and Signal Processing, Proc. of ISMM*, pages 311–320, 2000.