Playing with Kruskal: Algorithms for Morphological Trees in Edge-Weighted Graphs

Laurent Najman, Jean Cousty, and Benjamin Perret

Université Paris-Est, Laboratoire d'Informatique Gaspard-Monge, A3SI, ESIEE

Abstract. The goal of this paper is to provide linear or quasi-linear algorithms for producing some of the various trees used in mathemetical morphology, in particular the trees corresponding to hierarchies of watershed cuts and hierarchies of constrained connectivity. A specific binary tree, corresponding to an ordered version of the edges of the minimum spanning tree, is the key structure in this study, and is computed thanks to variations around Kruskal algorithm for minimum spanning tree.

1 Introduction

In the theoretical companion paper [1] of the present paper, we show how some morphological hierarchies [2–7] defined on an edge-weighted graph $G = (V, E, F)$ are related, and when it is possible, how they can be computed one from each other. In this paper, we provide efficient quasi-linear or linear algorithms to compute those hierarchies. In particular, this paper contains:

- provided that the edges are either already sorted or can be sorted in linear time, a quasi-linear $O(|E| \times \alpha(|V|))$ (where $\alpha()$ is the extremely slowly growing inverse of the single-valued Ackermann function) algorithm that computes a *binary partition tree by altitude ordering*, a fundamental structure that we post-process in the sequel;
- a linear $O(|V|)$ post-processing algorithm that computes the hierarchy of quasi-flat zones [1, 3] (also know as the α-tree [8]);
- a linear $O(|V|)$ post-processing algorithm that computes (hierarchies of) watershed cuts [9];
- a linear $O(|V|)$ post-processing algorithm that computes hierarchies by increasing attributes; as detailed here and in [1], such an algorithm can be used to obtain either constrained connectivity hierarchies [10] or watershed-based hierarchies [11].

To the best of our knowledge, the only published constrained connectivity algorithm, available in [10], has an unknown complexity and only computes one level of the hierarchy. An algorithm computing the whole hierarchy, relying on the component tree of the edge-weighted graph G, is roughly sketched in [12], but has a complexity higher than the one proposed in this paper and is less memory efficient. Concerning attribute-based hierarchies, the most efficient algorithm [13] has a complexity higher than the one proposed in this paper, and is less efficient.

C.L. Luengo Hendriks, G. Borgefors, R. Strand (Eds.): ISMM 2013, LNCS 7883, pp. 135–146, 2013.
© Springer-Verlag Berlin Heidelberg 2013

At the heart of our approach is the minimum spanning tree (MST): this tree T is a connected spanning graph of the graph G such that the weight of T: $F(T) := \sum_{e \in E(T)} F(e)$ is the least possible weight for a spanning graph of G. As detailed in section 2.1, we rely on Kruskal algorithm [14] for computing this MST. However, while producing the MST, we make use of another tree (detailed in section 2.2) that we call the *binary partition tree by altitude ordering.* Using Tarjan union-find (section 2.3), we propose in section 2.4 an efficient algorithm to compute this binary tree. Post-processing will be studied in section 3.

2 Binary Partition Tree and Minimum Spanning Tree

2.1 Kruskal Algorithm

Kruskal's algorithm [14] is a greedy algorithm that finds a minimum spanning tree for a connected weighted graph. It can be described as follows:

- create a forest $\mathcal{F}$ (a set of trees), where each vertex in the graph is a separate tree
- create a set S containing all the edges in the graph
- while S is nonempty and $\mathcal{F}$ is not yet a single tree
    - remove an edge with minimum weight from S
    - if that edge connects two different trees, then add it to the forest, combining two trees into a single tree
    - otherwise discard that edge.

At the termination of the algorithm, the forest has only one component and forms a minimum spanning tree of the graph.

The efficiency of Kruskal's algorithm relies on a disjoint-set data structure that keeps track of a set of elements partitioned into a number of disjoint (non-overlapping) subsets. The disjoint set problem consists in maintaining a collection $\mathcal{Q}$ of disjoint sets under the operation of union. Each set Q in $\mathcal{Q}$ is represented by a unique element of Q, called the *canonical element.* In the following, q_1 and q_2 denote two distinct elements. Three operations allow to manage the collection:

- `MakeSet`(q_1): add a new element q_1 to the collection $\mathcal{Q}$, provided that the element q_1 does not already belongs to a set in $\mathcal{Q}$.
- `FindCanonical`(q_1): return the canonical element of the set in $\mathcal{Q}$ which contains q_1.
- `Union`(q_1, q_2): let Q_1 and Q_2 be the two sets in $\mathcal{Q}$ whose canonical elements are q_1 and q_2 respectively (q_1 and q_2 must be different). Both sets are removed from $\mathcal{Q}$, their union $Q_3 = Q_1 \cup Q_2$ is added to $\mathcal{Q}$ and a canonical element for Q_3 is selected and returned.

An implementation of Kruskal algorithm is presented in Algorithm 1. In this implementation, we identify any element of V with an integer corresponding to its index in the finite set $|V|$. We save the edge of the MST in an array `MST`, hence obtaining a strict order on the edges; this order is necessary for post-processing.

Algorithm 1. Kruskal

```
Data: An edge-weighted graph (V, E, F).
Result: A minimum spanning tree MST
Result: A collection Q
  // Collection Q is initialized to ∅
1 e := 0;
2 for all x_i ∈ V do MakeSet(i);
3 for all edges {x, y} by (strict) increasing weight F({x, y}) do
4   c_x := Q.FindCanonical(x); c_y := Q.FindCanonical(y);
5   if c_x ≠ c_y then
6     Q.Union(c_x, c_y);
7     MST[e] := {x, y};
8     e := e + 1;
9   else DoSomething({x, y});
```

When computing an MST, the `DoSomething` procedure does nothing, *i.e.*, it discards the considered edge. In section 3.3, we show an example when the procedure `DoSomething` is useful.

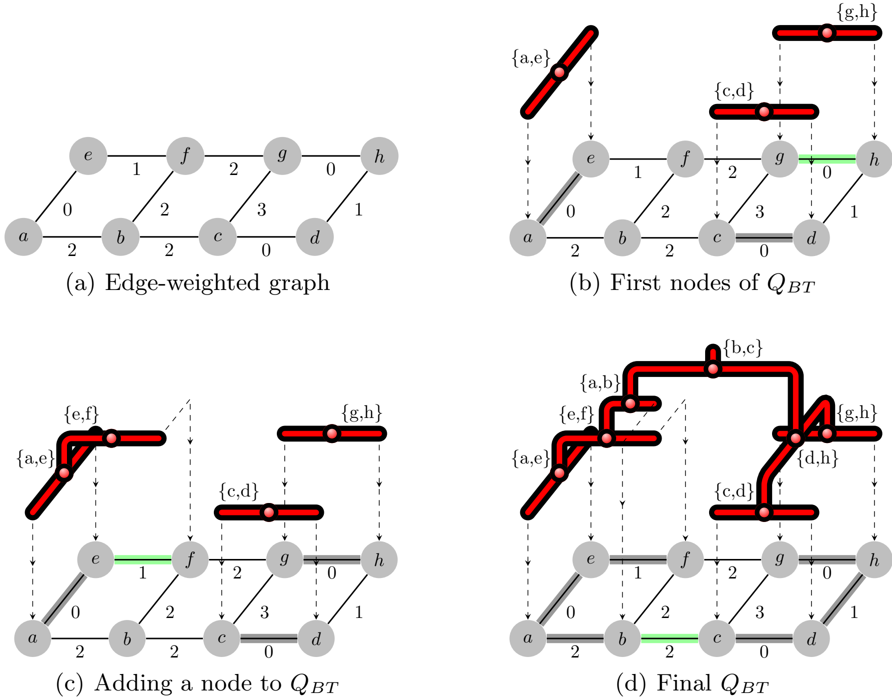
Procedure DoSomethingMST($\{x, y\}$)

```
// Ignore {x, y}
```

The main question in implementing Kruskal's algorithm is thus how to represent and implement the collection $\mathcal{Q}$. A good representation is to maintain the collection as a set of trees, *i.e* each element of $\mathcal{Q}$ is a tree. In the sequel of this paper, each set of the collection $\mathcal{Q}$ is represented by a rooted tree, where the canonical element of the set is the root of the tree. We are going to play with various tree representations of connected components. Kruskal's algorithm will not change, but different implementations for `Union` and `FindCanonical` will lead to two different trees, one of them being useful for connected filtering.

2.2 A Simple Algorithm That Yields a Binary Partition Tree by Altitude Ordering

The first tree we present is the useful one, although the proposed algorithm in this section is not an efficient one (we make a better proposal in section 2.4). As shown in [1] and in the sequel of this paper, post-processing this tree provides the min-tree of MST, the tree of quasi-flat zones and trees of watersheds. The main idea is the following: each time an edge $\{x, y\}$ is put into the MST, *i.e.* each time a union is made, we create a new node whose children are the two disjoint sets containing x and y. Intuitively, it is as if we break the edge in two, and add a node between the two points of the edge. The added node becomes the canonical element of the union of these two points (see Fig. 1.a, b, c and d).



(a) Edge-weighted graph (b) First nodes of Q_{BT}

(c) Adding a node to Q_{BT} (d) Final Q_{BT}

Fig. 1. A simple process for obtaining Q_{BT}, a binary tree providing a strict total order relation on the edges of the MST (see text)

Each node will thus either correspond to an edge of the MST or to a vertex of V. In the implementation, nodes of the tree are represented by integer: nodes corresponding to vertices are from 0 to $|V|-1$ and nodes corresponding to edges of the MST are from $|V|$ to $2|V|-2$. Such a numbering allows the tree itself to be provided thanks to an array `parent` that stores the parent of a given node. If node i does not have a parent, then $parent[i] := -1$. This algorithm is implemented in Q_{BT}.MakeSet, Q_{BT}.FindCanonical and Q_{BT}.Union.

Procedure Q_{BT}.MakeSet(q)

1 Q_{BT}.parent[q] $:= -1$; Q_{BT}.size += 1;

Function Q_{BT}.FindCanonical(q)

1 **while** Q_{BT}*.parent[q]*≥ 0 **do** $q :=$$Q_{BT}$.parent[$q$];
2 **return** q;

Function Q_{BT}.Union(c_x, c_y)

```
1 Q_BT.parent[c_x]:=Q_BT.size; Q_BT.parent[c_y]:=Q_BT.size;
2 Q_BT.MakeSet(Q_BT.size);
3 return Q_BT.size-1;
```

At the end, we obtain a collection Q_{BT} which is a binary tree. The Q_{BT} tree provides the order in which the edges have been put into the MST, the latest edge added to the tree being the root of the tree, *i.e.*, the highest one. In other words, the edges of the MST are strictly ordered by Q_{BT}, according to their altitude in the tree. We say that Q_{BT} is a *binary partition tree by altitude ordering* [1]. The complexity of Kruskal implemented with this specific union-find is in $O(|V|^2)$, and thus, this process is not very efficient.

2.3 Efficient MST Implementation with Tarjan Union-Find

Tarjan [15] proposed a very simple and very efficient algorithm called *union-find* to achieve any intermixed sequence of union and find. The implementation of this algorithm is given in procedure Q_T`.MakeSet` and functions Q_T`.Union` and Q_T`.FindCanonical`. To each element of the collection is associated a parent (as precedently) and a rank 'Rnk'. Both the mapping 'parent' and the mapping 'Rnk' are represented as arrays in memory. One of the two key heuristics to reduce the complexity is a technique called *path compression*, that was used by Tarjan to reduce the cost of `FindCanonical`. It consists, while searching for the root r of the tree which contains q, in considering each element p of the path from q to r (including q), and setting the parent of p 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 `Union` operation. The rank Rnk(c_x) of a canonical element c_x is a measure of the depth of the tree rooted in c_x, and is exactly the depth of this tree if the path compression technique is not used jointly with the union by rank technique. If the two canonical elements c_x and c_y have the same rank, then one of the elements, say c_y, is chosen arbitrarily to be the canonical element of the union: c_y becomes the parent of c_x; and the rank of c_y is incremented by one. 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 [15].

Procedure Q_T.MakeSet(q)

```
1 Q_T.parent[Q_T.size] := −1;  Q_T.Rnk[Q_T.size] := 0; Q_T.size += 1;
```

Function Q_T.FindCanonical(q)

```
1 r := q;
2 while Q_T.parent[r] ≥ 0 do r :=Q_T.parent[r];
3 while Q_T.parent[q] ≥ 0 do tmp := q; q :=Q_T.parent[q]; Q_T.parent[tmp] := r;
4 return r;
```

Function Q_T.Union(c_x, c_y)

1 **if** $(Q_T.Rnk[c_x] > Q_T.Rnk[c_y])$ **then** swap(c_x, c_y);
2 **if** $(Q_T.Rnk[c_x] == Q_T.Rnk[c_y])$ **then** Q_T.Rnk[c_y] += 1;
3 Q_T.parent[c_x] := c_y;
4 **return** c_y;

As stated at the beginning of this section, implementing Kruskal's algorithm with Tarjan Union-Find leads to a quasi-linear complexity: provided that the edges are either already sorted or can be sorted in linear time (for example with counting sort or radix sort), the complexity is $O(|E| \times \alpha(|V|))$, where $\alpha()$ is the extremely slowly growing inverse of the single-valued Ackermann function. Unfortunately, the tree built by Tarjan Union-Find is of no use for connected filtering: the path compression technique flattens the tree, and does not preserve the order by which the edges or the vertices are processed in the algorithm. In the next section, we are going to combine the two proposals into an efficient one.

2.4 An Efficient Algorithm That Yields a Binary Partition Tree by Altitude Ordering

The function Q_{BT}`.FindCanonical` is slow because it takes some time to find the canonical element for a connected component: we have to climb the tree until the root is found. We can use Tarjan Union-Find on Q_{BT} itself, *i.e.* we can use Q_T as a second collection that maintains a compressed representation of Q_{BT}, so that finding the canonical element is now in quasi-constant time. We also have to store the root of any tree in Q_{BT} in an array *Root* so that such a root can be found in constant time from any tree in Q_T. The implementation uses Q_{EBT}`.MakeSet`, Q_T`.FindCanonical` and Q_{EBT}`.Union`.

Procedure Q_{EBT}.MakeSet(q)

1 Q_{EBT}.Root[q]:=q; Q_{BT}.MakeSet(q); Q_T.MakeSet(q);

Function Q_{EBT}.Union(c_x, c_y)

1 t_u:=Q_{EBT}.Root[c_x]; t_v := Q_{EBT}.Root[c_y];
`// Union in` Q_{BT} `(without compression)`
2 Q_{BT}.parent[t_u] := Q_{BT}.parent[t_v] := Q_{BT}.size;
`// If children are needed, add them to the root`
3 Q_{BT}.children[Q_{BT}.size].add($\{t_u\}$); Q_{BT}.children[Q_{BT}.size].add($\{t_v\}$);
4 c:=Q_T.Union(c_x,c_y); `// Union in` Q_T `(with compression)`
5 Q_{EBT}.Root[c] := Q_{BT}.size; `// Update the root of` Q_{EBT}
6 Q_{BT}.MakeSet(Q_{BT}.size);
7 **return** Q_{BT}*.size-1*;

Function Q_{EBT}.FindCanonical(q)

1 **return** Q_T*.FindCanonical(q)*;

When Kruskal is finished, the tree Q_{BT} is exactly the same as the one in section 2.2. The only difference is thus the speed of the algorithm: thanks to the use of Tarjan Union-Find, the complexity of this Kruskal algorithm using Q_{EBT} is quasi-linear $O(|E| \times \alpha(|V|))$.

3 Post-Processing the Binary Tree

In this section, we are going to detail some linear $O(|V|)$ algorithms that produce, from Q_{BT}, (1) a watershed cut, (2) a hierarchy of quasi-flat zones, and (3) any attribute-based hierarchy (if the attribute is increasing).

As we have seen, each node of Q_{BT} corresponds either to a vertex of the graph or to an edge of the MST. Recall that edges of MST are sorted by a strict order relationship that follows increasing weight-edges: the $|V|$ first nodes of Q_{BT} are the vertices of $|V|$, and the root of Q_{BT} corresponds to the edge of the MST with the greatest weight. To ease the reading of the algorithms of this section, we provide below two helper functions that clarify how we can pass from the nodes of Q_{BT} to the edges of the MST and how to obtain the weight of the edge of the MST corresponding to a given node of Q_{BT}.

Function getEdge(n)

Data: a (non-leaf) node n of Q_{BT}
Result: the edge e of the MST corresponding to the nth node
1 **return** $n - |V|$;

Function weightNode(n)

Data: a (non-leaf) node of the tree
Result: the weight of the MST edge associated with the nth node of Q_{BT}
1 **return** *F(MST[getEdge(n)]);*

3.1 Watershed Cuts

A watershed cut [9] can be easily obtained in our framework. Indeed, those cuts are the "highest separations" of minima of the minimum spanning tree. We propose below a linear algorithm `watershed` that labels the edges of the MST with a flag stating whether or not an edge is a watershed edge. The main idea is to obtain the latest edge of the MST included in a given minimum in Q_{BT}. This can be done by counting the number of minima of the edge-weighted graph (G, F) thanks to a simple process than run through the nodes of Q_{BT} by increasing order, and checks whether or not a given node has an altitude lower than its parent and does not contain a minimum. We increment a counter for the ancestors of several minima. An edge is a watershed edge if it merges several catchment basins corresponding to different minima. The complexity of the function `watershed` is linear in the number of vertices, and thus the whole process that computes a watershed cut is quasi-linear.

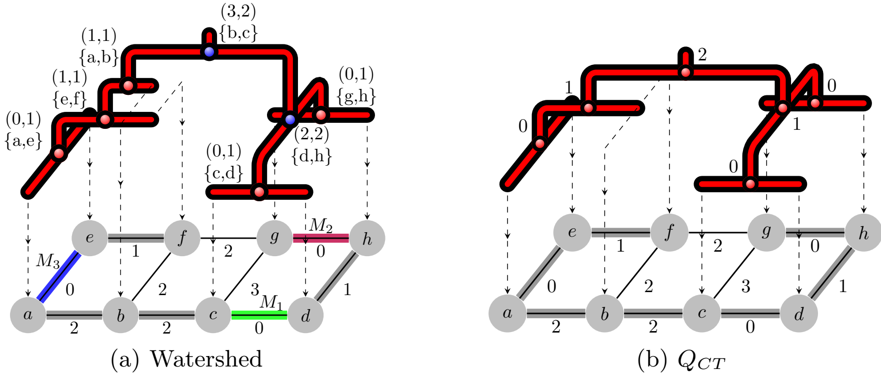


Fig. 2. Two trees derived from Q_{BT} by post-processing. (a) A hierarchical tree Q_{BT} on which watershed edges have a blue point. Edges M_1 (green), M_2 (red) and M_3 (blue) are the minima of the edge-weighted graph. (b) The tree of the quasi-flat zones hierarchy, a canonized version of Q_{BT} (*i.e.*, the min-tree of the MST.)

Function watershed

Data: Q_{BT}
Result: A binary array *ws* indicating which MST edges are watershed

```
1  for all leaf-nodes n of Q_BT do minima[n]:=0;
2  for each non-leaf node n of Q_BT by increasing order do
3      flag := TRUE; nb := 0;
4      for all c ∈ Q_BT.children[n] do
5          m := minima[c]; nb := nb + m;
6          if (m == 0) then flag := FALSE;
7      ws[getEdge(n)] := flag;
8      if (nb ≠ 0) then minima[n] := nb;
9      else
10         if (n is the root of Q_BT) then minima[n] := 1;
11         else
12             p := Q_BT.parent[n];
13             if (weightNode[n]<weightNode[p]) then minima[n] := 1;
14             else minima[n]:=0;
```

The set of watershed edges provides a MST of the neighborhood graph of the catchment basins [16]. In Fig. 2.a, the nodes of Q_{BT} corresponding to watershed edges have a blue point.

By removing from Q_{BT} all nodes that are not watershed ones, we obtain a filtered tree that corresponds to a hierarchy of watershed cuts, more precisely the one corresponding to an ultrametric watershed [12]. In the sequel of this paper, one can use indifferently either the original Q_{BT} or this modified one. In the

first case, one is working within the framework of the constrained connectivity, and in the second case, one is working within the framework of watershed cuts.

One of the interests of working with watershed cuts rather than with flat zones is that the hierarchy is smaller, as the super-pixels (catchment basins) provided by watershed cuts are larger than the super-pixels provided by the flat-zones. From a practical point of view, greater speed can thus be achieved thanks to watershed cuts.

3.2 Quasi-flat Zones Hierarchy

In this section, we post-process Q_{BT} to obtain the tree of the quasi-flat zones hierarchy [3], which is proved [1] to be Q_{CT}, the min-tree of the minimum spanning tree. The differences between Q_{BT} and Q_{CT} are illustrated in Fig. 1.d and 2.b. Q_{CT} can be computed directly thanks to a dedicated Union-Find procedure that we will describe in an extended version of this paper. Here, due to space constraints, we only propose a short post-processing that transforms Q_{BT} into the desired min-tree Q_{CT}. The implementation uses `Canonize`Q_{BT} to post-process Q_{BT}, and needs the children of a node.

Procedure CanonizeQ_{BT}

Data: Q_{BT}
Result: Q_{CT}, a canonized version of Q_{BT}

1 **for** *all nodes n of* Q_{BT} **do** Q_{CT}.parent[n]:=Q_{BT}.parent[n]; Q_{CT}.size+=1;
2 **for** *each non-leaf and non-root node n of* Q_{BT} *by decreasing order* **do**
3     $p := Q_{CT}$.parent[n];
4   **if** *(weightNode(p) == weightNode(n))* **then**
5   **for** *all* $c \in Q_{BT}$*.children*[n] **do** Q_{CT}.parent[c]:=p;
6   Q_{CT}.parent[n]:=n; `// Delete node` n `of` Q_{CT}

`// If needed, build the list of children`
7 **for** *all nodes n of* Q_{CT} **do**
8   p:=Q_{CT}.parent[n]; **if** $p \geq 0$ *and* $p \neq n$ **then** Q_{CT}.children[p].add(n);

The procedure `Canonize`Q_{BT} is in $O(|V|)$, and thus the whole process that computes a quasi-flat zones hierarchy is quasi-linear. To the best of our knowledge, this is the most efficient algorithm published for computing this hierarchy. However, for most image-processing tasks, the binary partition tree Q_{BT} can be used instead, and this is what we are going to do in the sequel of this paper.

3.3 Attribute-Based Hierarchies

Attributes It is easy to compute some attributes on each node of Q_{BT}: surface (number of vertices in a node), depth, volume or ordered markers are the most classical attributes [11]. Another attribute from the constrained connectivity framework [10] is the range. Any of those previous attributes are *increasing*: recall that an attribute A is increasing if when there is a parenthood relationship

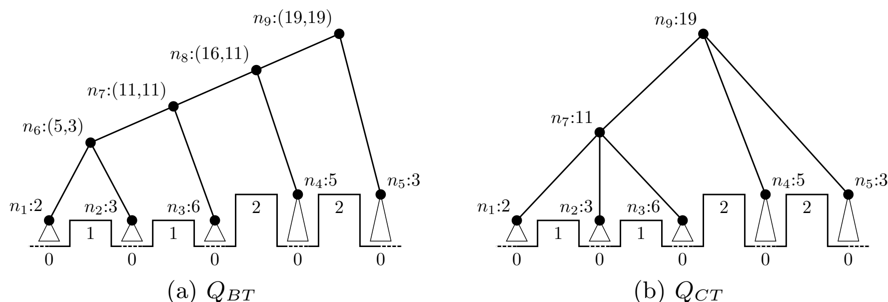
between two nodes n_1 and n_2, *i.e.* when the vertices of n_1 are contained in the vertices of n_2, then $A(n_1) \leq A(n_2)$. In the sequel of this section, we denote by `attributeComp`$[n]$ the attribute of the node n of Q_{BT}.

It is when computing an attribute that the procedure `DoSomethingMST` can be useful: for example, one can, using `DoSomethingMST`, register each edge of the graph at the correct place in Q_{BT}, by adding new nodes corresponding to these edges. Then, for example, we can imagine using an attribute such as the surface computed not from the vertices but from the edges.

Hierarchies. Attribute-based hierarchies are obtained by filtering the min-tree Q_{CT} of the MST, and computing the full hierarchy amounts to reweighting the MST. Intuitively, it is as if colored water fills up the branches of Q_{CT}, a merging of two components taking place when two different colors meet. The speed of the filling is controled by the attribute, *i.e.*, the branch is completely filled (*i.e.*, the branch is cut) when the amount of water is exactly equal to the attribute of the corresponding node of Q_{CT}. Some nodes exist in Q_{BT} but not in Q_{CT}. Those nodes correspond to an information not present in Q_{CT}: the order of processing, useful when a choice has to be made. A complete illustrative example explaining the process is detailed in Fig. 3. Computing the correct attribute is done thanks to the function `getAttribute`, that computes from `attributeComp` the attribute at the time of the merging. It consists in taking the highest attribute of all the children with the same original weight. The final weight of the MST can then be obtained by taking the lowest attribute of the two children of the node corresponding to the edge of the MST in Q_{BT}; this is done by `ComputeMergeAttributeMST`, the complexity of the whole post-processing being in $O(|V|)$. Once the re-weighted MST is computed, the hierarchical tree can be obtained by re-applying the algorithms of this paper. As seen in [1], the resulting hierarchy can be either a hierarchy of watershed cuts or a constrained-connectivity hierarchy.

Function getAttribute(n)

```
   Data: A node n of Q_BT
   Result: The attribute at the time of the merging
 1 if (n is the root) or (weightNode(parent[n]) ≠ weightNode(n)) then
 2 |   for all c children of n do getAttribute(c);
 3 |   attribute[n] := attributeComp[n];
 4 else
 5 |   max:=0;
 6 |   for all children c of n do
 7 |   |   v:=getAttribute(c);
 8 |   |   if v > max then max := v;
 9 |   attribute[n] := max;
10 return attribute[n];
```



(a) Q_{BT} (b) Q_{CT}

Fig. 3. Hierarchical trees Q_{BT} and Q_{CT}. At the bottom of (a) and (b) is the (edge-weighted) graph of a map with the respective weight of each edge. Attributes (in this example, the surface of a node) can easily be computed on either Q_{BT} or Q_{CT}. In (a), the numbers (k, l) in parenthesis represents respectively k=`attributeComp`[n] and l=`getAttribute`(n). In (a) and (b), the numbers k in n_i : k represents k=`attributeComp`[n]. When a flooding-by-attribute is performed on Q_{CT}, node n_1 disappears first at value 2, followed by node n_2 at value 3. When these the two corresponding branches are filled by water (cut from the tree), n_1 and n_2 are no longer minima. This is not the case for n_3, whose attribute is 6. When the corresponding branch is filled, n_3 stills marks a minimum, and stays so until the value 11, corresponding to node n_7. Thanks to the processing order embodied in Q_{BT}, `getAttribute` can compute the value at which a node disappears.

Procedure ComputeMergeAttributeMST

Data: Q_{BT}

Result: a reweighted MST G corresponding to the attribute-based hierarchy

```
1 for any non-leaf node n of Q_BT do
2     a_1 := attribute[children[n].left];
3     a_2 := attribute[children[n].right];
4     G[getEdge(n)] := min(a_1, a_2);
```

4 Conclusion

This paper has presented several elegant yet efficient algorithms for computing several morphological trees. At the heart of the processing is the minimum spanning tree, and in this paper we have proposed some variations on Kruskal algorithm. However, other approaches can be taken, and any other MST algorithm can be used first to produce a tree on which the algorithms of this paper can be applied. This would be needed in some situations, for example if the edges of the original graph do not fit in memory or if some parallel algorithm for minimum spanning tree is first needed. The unification theory provided in [1], together with the algorithms of this paper shed a new light on what has been done in mathematical morphology for a number of years, linking together some previously unrelated parts of the field.

Source code corresponding to this paper is available at
`http://www.esiee.fr/∼info/sm/`.

Acknowledgements. This work received funding from the Agence Nationale de la Recherche, contract ANR-2010-BLAN-0205-03 and through "Programme d'Investissements d'Avenir" (LabEx BEZOUT n°ANR-10-LABX-58).

References

1. Cousty, J., Najman, L., Perret, B.: Constructive links between some morphological hierarchies on edge-weighted graphs. In: Luengo Hendriks, C.L., Borgefors, G., Strand, R. (eds.) ISMM 2013. LNCS, vol. 7883, pp. 86–97. Springer, Heidelberg (2013)
2. Salembier, P., Oliveras, A., Garrido, L.: Anti-extensive connected operators for image and sequence processing. IEEE TIP 7(4), 555–570 (1998)
3. Nagao, M., Matsuyama, T., Ikeda, Y.: Region extraction and shape analysis in aerial photographs. CGIP 10(3), 195–223 (1979)
4. Meyer, F., Maragos, P.: Morphological scale-space representation with levelings. In: Nielsen, M., Johansen, P., Fogh Olsen, O., Weickert, J. (eds.) Scale-Space 1999. LNCS, vol. 1682, pp. 187–198. Springer, Heidelberg (1999)
5. Salembier, P., Garrido, L.: Binary partition tree as an efficient representation for image processing, segmentation, and information retrieval. IEEE TIP 9(4), 561–576 (2000)
6. Najman, L., Schmitt, M.: Geodesic saliency of watershed contours and hierarchical segmentation. IEEE PAMI 18(12), 1163–1173 (1996)
7. Morris, O.J., de Lee, M.J., Constantinides, A.G.: Graph theory for image analysis: an approach based on the shortest spanning tree. IEE Proc. on Communications, Radar and Signal 133(2), 146–152 (1986)
8. Ouzounis, G., Soille, P.: Pattern spectra from partition pyramids and hierarchies. In: Soille, P., Pesaresi, M., Ouzounis, G.K. (eds.) ISMM 2011. LNCS, vol. 6671, pp. 108–119. Springer, Heidelberg (2011)
9. Cousty, J., Bertrand, G., Najman, L., Couprie, M.: Watershed Cuts: Minimum Spanning Forests and the Drop of Water Principle. IEEE PAMI 31(8), 1362–1374 (2009)
10. Soille, P.: Constrained connectivity for hierarchical image partitioning and simplification. IEEE PAMI 30(7), 1132–1145 (2008)
11. Meyer, F., Najman, L.: Segmentation, minimum spanning tree and hierarchies. In: Najman, L., Talbot, H. (eds.) Mathematical Morphology: From Theory to Application, pp. 229–261. ISTE-Wiley, London (2010)
12. Najman, L.: On the equivalence between hierarchical segmentations and ultrametric watersheds. JMIV 40(3), 231–247 (2011) arXiv:1002.1887v2
13. Cousty, J., Najman, L.: Incremental algorithm for hierarchical minimum spanning forests and saliency of watershed cuts. In: Soille, P., Pesaresi, M., Ouzounis, G.K. (eds.) ISMM 2011. LNCS, vol. 6671, pp. 272–283. Springer, Heidelberg (2011)
14. Kruskal, J.B.: On the shortest spanning subtree of a graph and the traveling salesman problem. In: Proceedings of the AMS, vol. 7, pp. 48–50 (February 1956)
15. Tarjan, R.: Efficiency of a good but not linear set union algorithm. Journal of the ACM 22, 215–225 (1975)
16. Meyer, F.: Minimum spanning forests for morphological segmentation. In: ISMM, pp. 77–84 (September 1994)