A fair comparison of many max-tree computation algorithms *(Extended version of the paper submitted to ISMM 2013)*

Edwin Carlinet[1] and Thierry Géraud[1]

EPITA Research and Development Laboratory (LRDE)
`edwin.carlinet@lrde.epita.fr, thierry.geraud@lrde.epita.fr`

Abstract. With the development of connected filters for the last decade, many algorithms have been proposed to compute the max-tree. Max-tree allows to compute the most advanced connected operators in a simple way. However, no fair comparison of algorithms has been proposed yet and the choice of an algorithm over an other depends on many parameters. Since the need of fast algorithms is obvious for production code, we present an in depth comparison of five algorithms and some variations of them in a unique framework. Finally, a decision tree will be proposed to help user choose the right algorithm with respect to their data.

1 Introduction

In mathematical morphology, connected filters are those that modify an original signal by only removing connected components, hence those that preserve image contours. At the early stage, they were mostly used for image filtering [17, 14]. Breaking came from max and min-tree as hierarchical representations of connected components and from an efficient algorithm able to compute them [13]. Since then, usage of these trees has soared for more advanced forms of filtering: based on attributes [4], using new filtering strategies [13, 16], allowing new types of connectivity. They are also a base for other image representations, in [9] a tree of shapes is computed from a merge of min and max trees, in [21] a component tree is computed over the attributes values of the max-tree. Max-trees have been used in many applications: computer vision through motion extraction [13], features extraction with MSER [5], segmentation, 3D visualization [7]. With the increase of applications comes an increase of data type to process: 12-bit image in medical imagery [7], 16-bit or float image in astronomical imagery [2], and even multivariate data with special ordering relation [12]. With the improvement of optical sensors, images are getting bigger (so do image data sets) which urge the need of fast algorithms. Many algorithms have been proposed to compute the max-tree efficiently but only partial comparisons have been proposed. Moreover, some of them are dedicated to particular task (e.g., filtering) and are unusable for other purpose. We provide in this paper a full and fair comparison of state-of-the-art max-tree algorithms in a unique framework i.e. same architecture, same language (C++) and same outputs.

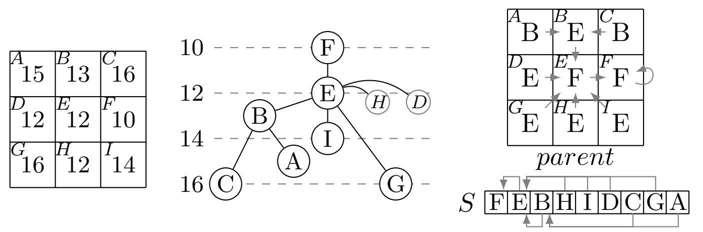
2 A tour of max-tree: definition, representation and algorithms

2.1 Basic notions for max-tree

Let $ima : \Omega \rightarrow V$ an image on regular domain Ω, having values on totally preordered set $(V, <)$ and let $\mathcal{N}$ a neighborhood on Ω. Let $\lambda \in V$, we note $[ima \leq \lambda]$ the set $\{p \in \Omega, ima(p) \leq \lambda\}$. Let $X \subset \Omega$, we note $CC(X) \subset \mathcal{P}(\Omega)$ the set of connected components of X w.r.t the neighborhood $\mathcal{N}$. $CC([ima = \lambda]), \lambda \in V\}$ are *level components* and $\{CC([ima \geq \lambda]), \lambda \in V\}$ (resp. $\leq$) is the set of upper components (resp. lower components). The latter endowed with the inclusion relation form a tree called the max-tree (resp. min-tree). The peak component of p at level λ noted P_p^λ is the upper component $X \in CC([ima \geq \lambda])$ such that $p \in X$.

2.2 Max-tree representation

Berger et al. [2] rely on a simple and effective encoding of component-trees using an image that stores the *parent* relationship that exists between components. A connected component is represented by a single point called the *canonical element* [2, 10] or *level root*. Let two points $p, q \in \Omega$, and p_r the root of the tree, we say that p is canonical if $p = p_r$ or $ima(parent(p)) < ima(p)$. A *parent* image shall verify the following three properties: 1) $parent(p) = p \Rightarrow p = p_r$ - the root points to itself and it is the only point verifying this property - 2) $ima(parent(p)) \leq ima(p)$ and 3) $parent(p)$ is canonical.



(a) Original image *ima* (b) Max-tree of *ima* (c) Max-tree representation

Fig. 1: Representation of a max-tree with a parent image and an array.

Furthermore, this representation requires an extra vector $S : \mathbb{N} \rightarrow \Omega$ of points that orders the nodes downward. Thus, S must verifies $\forall i, j \in \mathbb{N}\ i <$

$j \Rightarrow S[j] \neq \mathit{parent}(S[i])$. Ordering S vector allows to traverse the tree upward or downward without storing children of each node. Figure 1 shows an example of such a representation of a max-tree. This representation only requires $2.n.I$ bytes memory space where n is the number of pixels and I the size in bytes of an integer (points are positive offsets in a pixel buffer). The algorithms we compare have all been modified to output such a tree encoding.

2.3 Attribute filtering and reconstruction

A classical approach for object detection and filtering is to compute some features called attributes on max-tree nodes. An usual attribute is the number of pixels in components. Followed by a filtering, it leads to the well-known area opening. More advanced attributes have been used like elongation, moment of inertia [18] or even mumford-shah like energy [21].

Many max-tree algorithms only construct the *parent* image but do not care about S construction [19, 5], they output incomplete information. We require that max-tree algorithms give a "usable" tree, i.e., a tree that can be traversed upwards and downwards, that allows attribute computation and non-trivial filtering. The rational behind this requirement is that, for some applications, filtering parameters are not known yet when building the tree (e.g., for interactive visualization [7]). In the algorithms we compare in this paper, no attribute computation nor filtering are performed during tree construction for clarity reasons; yet they can be augmented to compute attribute and filtering at the same time. Algorithm 1 provides an implementation of attribute computation and direct-filtering with the representation. $\hat{f} : \Omega \times V \rightarrow \mathcal{A}$ is an application that projects a pixel p and its value $\mathit{ima}(p)$ in the attribute space $\mathcal{A}$. $\hat{+} : \mathcal{A} \times \mathcal{A} \rightarrow \mathcal{A}$ is an associative operator used to merge attributes of different nodes. COMPUTE-ATTRIBUTE starts with computing attribute of each singleton node and merge them from leaves toward root. Note that this simple code relies on the fact that a node receives all information from its children before passing its attribute to the parent. Without any ordering on S, it would not have been possible. DIRECT-FILTER is an implementation of direct filtering as explained in [13] that keeps all nodes passing a criterion λ and lowers nodes that fails to the last ancestor "alive". This implementation has to be compared with the one in [19] that only uses *parent*. This one is shorter, faster and clearer above all.

3 Max-tree algorithms

Max-tree algorithms can be classified in three classes:

Immersion algorithms. It starts with building N disjoints singleton for each pixel and sort them according to their gray value. Then, disjoint sets merge to form a tree using Tarjan's union-find algorithm [15].

Flooding algorithms. A first scan allows to retrieve the root which is a pixel at lowest level in the image. Then, it performs a propagation by flooding firsts the neighbor at highest level i.e. a depth first propagation. [13, 20].

Algorithm 1 Computation of attributes and filtering.

```
function
COMPUTE-ATTRIBUTE($S$, $parent$, $ima$)
    $p_{root} \leftarrow S[0]$
    for all $p \in S$ do
        $attr(p) \leftarrow \hat{f}(p, ima(p))$
    for all $p \in S$ backward, $p \neq p_{root}$ do
        $q \leftarrow parent(p)$
        $attr(q) \leftarrow attr(q) \hat{+} attr(p)$
    return $attr$
```

```
function
DIRECT-FILTER($S$, $parent$, $ima$, $attr$)
    $p_{root} \leftarrow S[0]$
    if $attr(p_{root}) < \lambda$ then $out(p_{root}) = 0$
    else $out(p_{root}) = ima(p_{root})$
    for all $p \in S$ forward do
        $q \leftarrow parent(p)$
        if $ima(q) = ima(p)$ then
            $out(p) = out(q)$    ▷ $p$ not canonical
        else if $attr(p) < \lambda$ then
            $out(p) \leftarrow out(q)$    ▷ Criterion failed
        else
            $out(p) \leftarrow ima(p)$    ▷ Criterion pass
    return $out$
```

Merge-based algorithms. They divide an image in blocks and compute the max-tree on each sub-image using another max-tree algorithm. Sub max-trees are then merged to form the tree of the whole image. Those algorithms are well-suited for parallelism using a *map-reduce* approach [19]. When blocks are image lines, a dedicated 1D max-tree algorithm can be used [6, 8].

3.1 Immersion algorithms

Berger et al. [2] and Najman and Couprie [10] proposed two algorithms based on Tarjan's union-find. They consist in tracking disjoints connected components and merge them in bottom up fashion. First, pixels are sorted in an array S where each pixel p represent the singleton set $\{p\}$. Then, we process pixels of S in backward order. When a pixel p is processed, it looks for already processed neighbor ($\mathcal{N}(p)$) and merges with neighboring connected components to form a new connected set rooted in p. The merging process consists in updating the *parent* pointer of neighboring component roots toward p. Thus, the union-find relies on three processes:

– `make-set(parent, x)` that builds the singleton set $\{x\}$,
– `find-root(parent, x)` that finds the root of the component that contains x,
– `merge-set(parent, x, y)` that merges components rooted in x and y and set x as the new root.

Based on the above functions, a simple max-tree algorithm is given below:

```
procedure MAXTREE($ima$)
    $S \leftarrow$ sorts pixels increasing
    for all $p \in S$ backward do
        make-set($parent, p$)
        for all $n \in \mathcal{N}_p$ processed do
        $r \leftarrow$ find-root$(parent, n)$
        **if** $r \neq p$ **then**
            merge-set$(parent, p, r)$
```

`find-root` is a $O(n)$ function that makes the above procedure a $O(n^2)$ algorithm. Tarjan [15] discussed two important optimizations for his algorithm to avoid a quadratic complexity:

- Root path compression. When *parent* is traversed to get the root of the component, points of the path used to find the root collapse to the actual root the component. However, path compression should not be applied on *parent* image because it removes the hierarchical structure of the tree. As consequence, we apply path compression on an intermediate image *zpar* that stores the root of disjoints components. Path compression bounds union-find complexity to $O(n \log n)$ and has been applied in [2] and [10].
- Union-by-rank. When merging two components A and B, we have to select one the roots to represent the newly created component. If A has a *rank* greater than B then $root_A$ is selected as the new root, $root_B$ otherwise. When rank matches the depth of trees, it enables tree balancing and guaranties a $O(n \log n)$ complexity for union-find. When used with path compression, it allows to compute the max-tree in quasi-linear time ($O(n.\alpha(n))$ where $\alpha(n)$ is the inverse of Ackermann function which is very low-growing). Union-by-rank has been applied in [10].

Note that *parent* and *zpar* encode two different things, *parent* encodes the max tree while *zpar* tracks disjoints set of points and also use a tree. Thus, union-by-rank and root path compression shall always be applied on *zpar* but never on *parent*.

Algorithm 2 is the union-find based max-tree algorithm as proposed by Berger et al. [2]. It starts with sorting pixels that can be done with a counting sort algorithm for low-quantized data or with a radix sort-based algorithm for high quantized data[1].Then it annotates all pixels as *unprocessed* with -1 (in standard implementations pixel are positive offsets in a pixel buffer). Later in the algorithm, when a pixel p is processed it becomes the root of the component i.e $parent(p) = p$ with $p \neq -1$, thus testing $parent(p) \neq -1$ stands for *is p already processed*. Since S is processed in reverse order and `merge-set` sets the root of the tree to the current pixel p ($parent(r) \leftarrow p$), it ensures that the parent p will be seen before its child r when traversing S in the direct order.

Union-by-rank Algorithm 3 is similar to algorithm 2 but augmented with union-by-rank. It first introduces a new image *rank*. The `make-set` step creates a tree with a single node, thus with a rank set to 0. The *rank* image is then used when merging two connected set in *zpar*. Let z_p the root of the connected component of p, and z_n the root of connected component of $n \in \mathcal{N}(p)$. When merging two components, we have to decide which of z_p or z_n becomes the new root w.r.t their rank. If $rank(z_p) < rank(z_n)$, z_p becomes the root, z_n otherwise. If both z_p and z_n have the same rank then we can choose either z_p or z_n as the

Algorithm 2 Union find without union-by-rank

```
function FIND-ROOT($par$, $p$)
    if $par(p) \neq p$ then $par(p) \leftarrow$ FIND-ROOT($par, par(p)$)
    return $par(p)$
function MAXTREE($ima$)
    for all $p$ do $parent(p) \leftarrow -1$
    $S \leftarrow$ sorts pixels increasing
    for all $p \in S$ backward do
        $parent(p) \leftarrow p$; $zpar(p) \leftarrow p$                    ▷ make-set
        for all $n \in \mathcal{N}_p$ such that $parent(n) \neq -1$ do
            $r \leftarrow$ FIND-ROOT($zpar, n$)
            if $r \neq p$ then
                $zpar(r) \leftarrow p$; $parent(r) \leftarrow p$            ▷ merge-set
    CANONIZE($parent$, $S$)
    return ($parent, S$)
```

new root, but the rank should be incremented by one. On the other hand, the relation *parent* is unaffected by the union-by-rank, p becomes the new root whatever the rank of z_p and z_n. Whereas without balancing the root of any point p in *zpar* matches the root of p in parent, this is not the case anymore. For every connected components we have to keep a connection between the root of the component in *zpar* and the root of max-tree in *parent*. Thus, we introduce an new image *repr* that keeps this connection updated.

The union-by-rank technique and structure update are illustrated in Figure 2. The algorithm has been running until processing E at level 12, the first neighbor B has already been treated and neighbors D and F are skipped because not yet processed. Thus, the algorithm is going to process the last neighbor H. z_p is the root of p in *zpar* and we retrieve the root z_n of n with `find-root` procedure. Using *repr* mapping, we look up the corresponding point r of z_n in *parent*. The tree rooted in r is then merged to the tree rooted in p (*parent* merge). Back in *zpar*, components rooted in z_p and z_n merge. Since they have the same rank, we choose arbitrary z_p to be the new root.

Algorithm 3 is slightly different from the one of Najman and Couprie [10]. They use two union-find structure, one to build the tree, the other to handle flat zones. In their paper, `lowernode[`z_p`]` is an array that maps the root of a component z_p in *zpar* to a point of current level component in *parent* (just like `repr(`z_p`)` in our algorithm). Thus, they apply a second union-find to retrieve the canonical. This extra union-find can be avoided because `lowernode[x]` is already a canonical element, thus $findoot$ on $lowernode(z_p)$ is useless and so does *parent* balancing on flat zones.

Canonization Both algorithms call the CANONIZE(p)rocedure to ensure that any node's parent is a canonical node. In algorithm 4, canonical property is broadcast downward. S is traversed in direct order such that when process-

Algorithm 3 Union find with union-by-rank

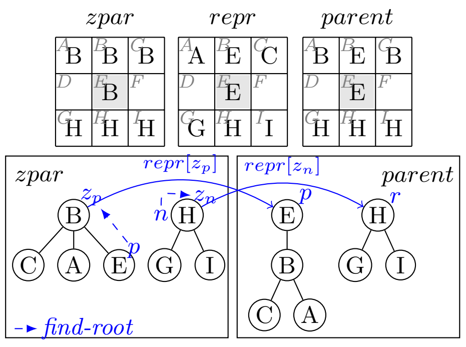
```
procedure MAXTREE($ima$)
    for all $p$ do $parent(p) \leftarrow -1$
    $S \leftarrow$ sorts pixels increasing
    for all $p \in S$ backward do
        $parent(p) \leftarrow p$; $zpar(p) \leftarrow p$                          ▷ make-set
        $rank(p) \leftarrow 0$; $repr(p) \leftarrow p$
        $z_p \leftarrow p$
        for all $n \in \mathcal{N}_p$ such that $parent(n) \neq -1$ do
            $z_n \leftarrow$ FIND-ROOT($zpar, n$)
            if $z_n \neq z_p$ then
                $parent(repr(z_n)) \leftarrow p$
                if $rank(z_p) < rank(z_n)$ then $swap(z_p, z_n)$
                $zpar(z_n) \leftarrow z_p$                                   ▷ merge-set
                $repr(z_p) \leftarrow p$
                if $rank(z_p) = rank(z_n)$ then
                    $rank(z_p) \leftarrow rank(z_p) + 1$
    CANONIZE($parent$, $S$)
    return ($parent, S$)
```

ing a pixel p, its parent q has the canonical property that is $parent(q)$ is a canonical element. Hence, if q and $parent(q)$ belongs to the same node i.e $ima(q) = ima(parent(q))$, the parent of p is set to the component's canonical element: $parent(q)$.

Algorithm 4 Canonization algorithm

```
procedure CANONIZE($ima$, $parent$, $S$)
    for all $p$ in $S$ forward do
        $q \leftarrow parent(p)$
        if $ima(q) = ima(parent(q))$ then
            $parent(p) \leftarrow parent(q)$
```

Level compression Union-by-rank provides time complexity guaranties at the price of extra memory requirement. When dealing with huge images this results in a significant drawback (e.g. RAM overflow...). Since the last point processed always becomes the root, union-find without rank technique tends to create degenerated tree in flat zones. Level compression avoids this behavior by a special handling of flat zones. In algorithm 5, p is the point in process at level $\lambda = ima(p)$, n a neighbor of p already processed, z_p the root of P_p^λ (at first $z_p = p$), z_n the root of P_n^λ. We suppose $ima(z_p) = ima(z_n)$, thus z_p and z_n belong to the same node and we can choose any of them as a canonical element. Normally p should become the root with child z_n but level compression inverts the relation,


(a)

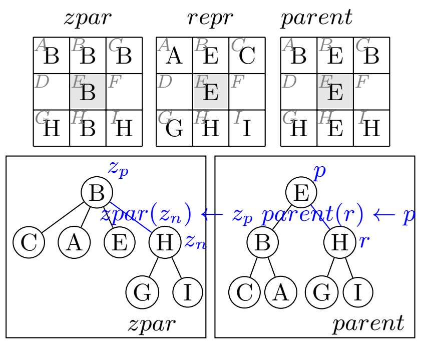

(b)

Fig. 2: Union-by-rank. (a) State of the algorithm before processing the neighbor H from E. (b) State of the algorithm after processing.

z_n is kept as the root and z_p becomes a child. Since *parent* may be inverted, S array is not valid anymore. Hence S is reconstructed, as soon as a point p gets attached to a root node, p will be not be processed anymore so its inserted in back of S. At the end S only misses the tree root which is $parent[S[0]]$.

3.2 Flooding algorithms

Salembier et al. [13] proposed the first efficient algorithm to compute the max-tree. A propagation starts from the root that is the pixel at lowest level l_{min}. Pixels in the propagation front are stored in a hierarchical queue that allows a direct access to pixels at a given level in the queue. The `flood(`λ`, `r`)` procedure (see algorithm 6) is in charge of flooding the peak component P_r^λ and building the corresponding sub max-tree rooted in r. It proceeds as follows: first pixels at level λ are retrieved from the queue, their *parent* pointer is set to the canonical element r and their neighbors n are analyzed. If n is not in queue and has not yet been processed, then n is pushed in the queue for further process sing and n is marked as processed ($parent(n)$ is set to `INQUEUE` which is any value different from -1). If the level l of n is higher than λ then n is in the childhood of the current node, thus `flood` is called recursively to flood the peak component P_n^l rooted in n. During the recursive flood, some points can be pushed in queue between level λ and l. Hence, when `flood` ends, it returns the level l' of n's parent. If $l' > \lambda$, we need to flood level l' until $l' \leq \lambda$ i.e until there are no more points in the queue above λ. Once all pixels at level λ have processed, we need to retrieve the level *lpar* of parent component and attach r to its canonical element. A *levroot* array stores canonical element of each level component and -1 if the component is empty. Thus we just have to traverse *levroot* looking for $lpar = \max\{h < \lambda, levroot[h] \neq -1\}$ and set the parent of r to $levroot[lpar]$.

Algorithm 5 Union find with level compression

```
function MAXTREE($ima$)
    for all $p$ do $parent(p) \leftarrow -1$
    $S \leftarrow$ sorts pixels increasing
    $j = N - 1$
    for all $p \in S$ backward do
        $parent(p) \leftarrow p$; $zpar(p) \leftarrow p$                ▷ make-set
        $z_p = p$
        for all $n \in \mathcal{N}_p$ such that $parent(n) \neq -1$ do
            $z_n \leftarrow$ FIND-ROOT$(zpar, n)$
            if $z_p \neq z_n$ then
                if $ima(z_p) = ima(z_n)$ then SWAP$(()z_p, z_n)$
                $zpar(z_n) \leftarrow z_p$; $parent(z_n) \leftarrow z_p$          ▷ merge-set
                $S[j] \leftarrow z_n$; $j \leftarrow j - 1$
    $S[0] \leftarrow parent[S[0]]$
    CANONIZE($parent$, $S$)
    return ($parent, S$)
```

Since the construction of *parent* is bottom-up, we can safely insert p in front of the S array each time $parent(p)$ is set. For a level component, the canonical element is the last element inserted ensuring a correct ordering of S. Note that the first that gets a the minimum level of the image is not necessary. Instead, we could have called `flood` in `Max-tree` procedure until the parent level returned by the function was -1, i.e the last flood call was processing the root. Anyway, this pass has other advantages for optimization that will be discussed in the implementation details section.

Salembier et al. [13]'s algorithm was rewritten in a non-recursive implementation in Hesselink [3] and later by Nistér and Stewénius [11] and Wilkinson [20]. These algorithms differ in only two points. First, [20] uses a pass to retrieve the root before flooding to mimics the original recursive version while Nistér and Stewénius [11] does not. Second, priority queues in [11] use an unacknowledged implementation of heap based on hierarchical queues while in [20] they are implemented using a standard heap (based on comparisons). The algorithm 7 is a code transcription of the method described in Nistér and Stewénius [11]. The array *levroot* in the recursive version is replaced by a stack with the same purpose: storing the canonical element of level components. The hierarchical queue *hqueue* is replaced by a priority queue *pqueue* that stores the propagation front. The algorithm starts with some initialization and choose a random point p_{start} as the flooding point. p_{start} is enqueued and pushed on *levroot* as canonical element. During the flooding, the algorithm picks the point p at highest level (with the highest priority) in the queue, and the canonical element r of its component which is the top of *levroot* (p is not removed from the queue). Like in the recursive version, we look for neighbors n of p and enqueue those that have not yet been seen. If $ima(n) > ima(p)$, n is pushed on the stack and we immediately flood n (a *goto* that mimics the recursive call). On the other hand, if all neigh-

Algorithm 6 Salembier et al. [13] max-tree algorithm

```
function FLOOD($\lambda$, $r$)
    while $hqueue[\lambda]$ not empty do
        $p \leftarrow$ POP($hqueue[\lambda]$)
        $parent(p) \leftarrow r$
        if $p \neq r$ then INSERT_FRONT($S$, $p$)
        for all $n \in \mathcal{N}(p)$ such that $parent(p) = -1$ do
            $l \leftarrow ima(n)$
            if $levroot[l] = -1$ then $levroot[l] \leftarrow n$
            PUSH($hqueue[l], n$)
            $parent(n) \leftarrow$ INQUEUE
            while $l > \lambda$ do
                $l \leftarrow flood(l, levroot[l])$
                                                            ▷ Attach to parent
    $levroot[\lambda] \leftarrow -1$
    $lpar \leftarrow \lambda - 1$
    while $lpar \geq 0$ and $levroot[lpar] = -1$ do
        $lpar \leftarrow lpar - 1$
    if $lpar \neq -1$ then
        $parent(r) \leftarrow levroot[lpar]$
    INSERT_FRONT($S$, $r$)
    return $lpar$
```

bors are in the queue or already processed then p is *done*, it is removed from the queue, $parent(p)$ is set its the canonical element r and if $r \neq p$, p is added to S (we have to ensure that the canonical element will be inserted last). Once p removed from the queue, we have to check if the level component has been fully processed in order to attach the canonical element r to its parent. If the next pixel q has a different level than p, we call the procedure `ProcessStack` that pops the stack, sets parent relationship between canonical elements and insert them in S until the top component has a level no greater than $ima(q)$. If the stack top's level matches q's level, q extends the component so no more process is needed. On the other hand, if the stack gets empty or the top level is lesser

Algorithm 6 Salembier maxtree algorithm (continued)

```
function MAX-TREE(ima)
    for all $h$ do $levroot[h] \leftarrow -1$
    for all $p$ do $parent(p) \leftarrow -1$
    $l_{min} \leftarrow \min_p ima(p)$
    $p_{min} \leftarrow \arg\min_p ima(p)$
    PUSH($hqueue[l_{min}]$, $p_{min}$)
    $levroot[lmin] \leftarrow p_{min}$
    FLOOD($l_{min}$, $p_{min}$)
```

than $ima(q)$, then q is pushed on the stack as the canonical element of a new component. The algorithm ends when all points in queue have been processed, then S only misses the root of the tree which is the single element that remains on the stack.

Algorithm 7 Non-recursive max-tree algorithm [11, 20]

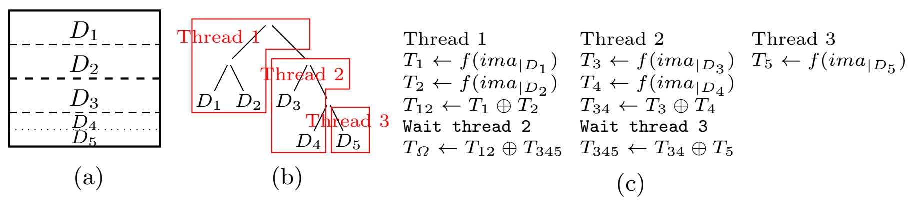
```
function MAX-TREE($ima$)
    for all $p$ do $parent(p) \leftarrow -1$
    $p_{start} \leftarrow$ any point in $\Omega$
    PUSH($pqueue, p_{start}$); PUSH($levroot, p_{start}$)
    $parent(p_{start}) \leftarrow$ INQUEUE
    loop
        $p \leftarrow$ TOP($pqueue$); $r \leftarrow$ TOP($levroot$)
        for all $n \in \mathcal{N}(p)$ such that $parent(p) = -1$ do
            PUSH($pqueue, n$)
            $parent(n) \leftarrow$ INQUEUE
            if $ima(p) < ima(n)$ then
                PUSH($levroot, n$)
                goto 7
        { $p$ is done }
        POP($pqueue$)
        $parent(p) \leftarrow r$
        if $p \neq r$ then INSERT_FRONT($S$, $p$)
    while $pqueue$ not empty do;
        { all points at current level done ? }
        $q \leftarrow$ TOP($pqueue$)
        if $ima(q) \neq ima(r)$ then                    ▷ Attach $r$ to its parent
            PROCESSSTACK($r$, $q$)
    repeat
    $root \leftarrow$ POP($levroot$)
    INSERT_FRONT($S$, $root$)
```

Algorithm 7 Non-recursive max-tree algorithm (continued)

```
procedure ProcessStack($r$, $q$)
    $\lambda \leftarrow ima(q)$
    pop($levroot$)
    while $levroot$ not empty and $\lambda < ima(\text{top}(levroot))$ do
        insert_front($S$, $r$)
        $r \leftarrow parent(r) \leftarrow \text{pop}(levroot))$
    if $levroot$ empty or $ima(\text{top}(levroot)) \neq \lambda$ then
        push($levroot, q$)
    $parent(r) \leftarrow \text{top}(levroot)$
    insert_front($S$, $r$)
```

3.3 Merge-based algorithms and parallelism

Merge-based algorithms consist in computing max-tree on sub-parts of images and merging back trees to get the max-tree of the whole image. Those algorithms are typically well-suited for parallelism since they adopt a map-reduce idiom. Computation of sub max-trees (map step), done by any sequential method and merge (reduce-step) are executed in parallel by several threads. In order to improve cache coherence, images should be split in contiguous memory blocks that is, splitting along the first dimension if images are row-major. Figure 3 shows an example of parallel processing using map-reduce idiom. Choosing the right number of splits and jobs distribution between threads is a difficult topic that depends on the architecture (number of threads available, power frequency of each core). If the domain is not split enough (number of chunks no greater than number of threads) the parallelism is not maximal, some threads become idle once they have done their jobs, or wait for other thread to merge. On the other hand, if the number of split gets too large, merging and thread synchronization cause significant overheads. Since work balancing and thread management are outside the current topic, they are delayed to high level parallelism library such as TBB.



Thread 1	Thread 2	Thread 3
$T_1 \leftarrow f(ima_{\mid D_1})$	$T_3 \leftarrow f(ima_{\mid D_3})$	$T_5 \leftarrow f(ima_{\mid D_5})$
$T_2 \leftarrow f(ima_{\mid D_2})$	$T_4 \leftarrow f(ima_{\mid D_4})$	
$T_{12} \leftarrow T_1 \oplus T_2$	$T_{34} \leftarrow T_3 \oplus T_4$	
Wait thread 2	Wait thread 3	
$T_\Omega \leftarrow T_{12} \oplus T_{345}$	$T_{345} \leftarrow T_{34} \oplus T_5$	

Fig. 3: Map-reduce idiom for max-tree computation. (a) Sub-domains of *ima*. (b) A possible distribution of jobs by threads. (c) Map-reduce operations. f is the map operator, $\oplus$ the merge operator.

The procedure in charge of merging sub-trees T_i and T_j of two adjacent domains D_i and D_j is given in algorithm 8. For two neighbors p and q in the junction of D_i, D_j, it connects components of p's branch in T_i to components of q's branch in T_j until a common ancestor is found. Let x and y, canonical elements of components to merge with $ima(x) \geq ima(y)$ (x is in the childhood to y) and z, canonical element of the parent component of x. If x is the root of the sub-tree then it gets attached to y and the procedure ends. Otherwise, we traverse up the branch of x to find the component that will be attached to y that is the lowest node having a level greater than $ima(y)$. Once found, x gets attached to y, and we now have to connect y to x's old parent. Function `findrepr(p)` is used to get the canonical element of p's component whenever the algorithm needs it.

Algorithm 8 Tree merge algorithm

```
function FINDREPR($par$, $p$)
    if $ima(p) \neq ima(par(p))$ then return $p$
    $par(p) \leftarrow$ FINDREPR($par, par(p)$)
    return $par(p)$

procedure CONNECT(p,q)
    $x \leftarrow$ FINDREPR($parent, p$)
    $y \leftarrow$ FINDREPR($parent, q$)
    if $ima(x) < ima(y)$ then SWAP($x, y$)
    while $x \neq y$ do                                  ▷ common ancestor found ?
        $parent(x) \leftarrow$ FINDREPR($parent, parent(x)$);
        $z \leftarrow parent(x)$
        if $x = z$ then                                  ▷ $x$ is root
            $parent(x) \leftarrow y$; $y \leftarrow x$
        else if $ima(z) \geq ima(y)$ then
            $x \leftarrow z$
        else
            $parent(x) \leftarrow y$
            $x \leftarrow y$
            $y \leftarrow z$

procedure MERGETREE($D_i$, $D_j$)
    for all $(p, q) \in D_i \times D_j$ such that $q \in \mathcal{N}(p)$ do
        CONNECT($p$,$q$)
```

Once sub-trees have been computed and merged into a single tree, it does not hold canonical property (because non-canonical elements are not updated during merge). Also, reduction step does not merge S array corresponding to sub-trees (it would imply reordering S which is more costly than just recomputing it at the end). Algorithm 9 performs canonization and reconstructs S array from *parent* image. It uses an auxiliary image *dejavu* to track nodes that have already been inserted in S. As opposed to other max-tree algorithms, construction of S and processing of nodes are top-down. For any points p, we traverse in a recursive way its path to the root to process its ancestors. When the recursive call returns, $parent(p)$ is already inserted in S and holds the canonical property, thus we can safely insert back p in S and canonize p as in algorithm 4.

Algorithm 9 Canonization and S computation algorithm

```
procedure CanonizeRec(p)
    $dejavu(p) = true$
    $q \leftarrow parent(p)$
    if not $dejavu(q)$ then                              ▷ Process parent before $p$
        CanonizeRec(q)
    if $ima(q) = ima(parent(q)$ then                      ▷ Canonize
        $parent(p) \leftarrow parent(q)$
    InsertBack($S, p$)

for all $p$ do $dejavu(p) \leftarrow False$
for all $p \in \Omega$ such that not $dejavu(p)$ do
    CanonizeRec($p$)
```

3.4 Implementation details

Algorithms have been implemented in pure C++ using STL implementation of some basic data structures (heaps, priority queues), Milena image processing library to provide fundamental image types and I/O functionality, and Intel TBB for parallelism. Specific implementation optimizations are listed below:

- Sort optimization. Counting sort is used when quantization is lower than 18 bits. For large integer of q bits, it switches to $2^1 6$-based radix sort requiring $q/16$ counting sort.
- Pre-allocation. Queues and stacks... are pre-allocated to avoid dynamic memory reallocation. Hierarchical queues are also pre-allocated by computing image histogram as a pre-processing.
- Priority-queues. Heap is implemented with hierarchical queues when quantization is less than 18 bits. For large integer it switches to the STL standard heap implementation. A y-fast trie can be used for large integer ensuring a better complexity (see subsection 3.5) but no performance gain has been obtained.
- Map-reduce. In parallel version of algorithms, all instructions that deals about S construction and *parent* canonization have been removed since they are S is reconstructed from scratch and *parent* canonized by procedure 9

3.5 Complexity analysis

Let $n = H * W$ with H the image height, W the image width and n the total number of pixels. Let k, the number of values in V.

- Immersion-based algorithms requires sorting pixels which has a complexity of $\Theta(n+k)$ $(k \ll n)$ for small integers (counting sort), $O(n \log \log n)$ for large integers (hybrid radix sort), and $O(n \log n)$ for generic data type with a more complicated ordering relation (comparison sort). The union-find is $O(n \log n)$ and $O(n\alpha(n))$ when used with union-by-rank. [1]. The canonization step is linear and does not use extra memory. Memory-wise, sorting may require an auxiliary buffer depending on the algorithm and histograms for integer sorts so $\Theta(n+k)$. Union without rank requires a *zpar* image for path compression $(\Theta(n))$ and the system stack for recursive call in `findroot` which is $O(n)$ (`findroot` could be non-recursive, but memory space is saved at cost of an higher computational time). When union-by-rank is used, it requires two extra images (*rank* and *repr*) of n pixels each.
- Flooding-based algorithms require a heap or hierarchical queues to retrieve the highest point in the propagation queue. Each point is inserted once and removed once (however they may be visited more than once in non-recursive versions) thus the complexity is $\Theta(n.p)$ where p is the cost of pushing or popping from the heap. If the heap is encoded with a hierarchical queue as in [13] or [11], it uses $n + 2.k$ memory space, insertion is $O(1)$, access to the maximum is $O(1)$ but popping is $O(k)$ (in the recursive version, we loop on *levroot* to get the parent level). In practice, in images with small integers, gray level difference between neighboring pixels is far to be as large as k. With high dynamic image, the heap can be implemented as a y-fast trie, which has insertion and deletion in $O(\log \log k)$ and access to maximum element in $O(1)$. For any other data type V, a "standard" heap based on comparisons requires n extra space, allows insertion and deletion in $O(\log n)$ and has a constant access to maximal element. Those algorithms need also an array or a stack *levroot* to store canonical elements of respective size k and n. Salembier's algorithm uses the system stack for a recursion of maximum depth k, hence $O(k)$.
- For merge-based algorithm, complexity is a bit-harder to compute. Let $\mathcal{A}(k, n)$ the complexity of the underlying algorithm used to compute max-tree on sub-domains and $s = 2^h$ the number of sub-domains. Map-reduce algorithm requires s mapping operations and $s - 1$ merges. A good map-reduce algorithm would lead split domain to form a full and complete tree so we assume all leaves to be at level h. Merging sub-trees of size $n/2$ has been analyzed in [19] and is $O(k \log n)$ (we merge nodes of every k levels using union-find without union-by-rank). Thus, the complexity of the reduction is $O(Wk \log n)$

[1] $\alpha(n)$, the inverse of Ackermann function, is very low growing, $\alpha(10^{80}) \simeq 4$

and the complexity S as a function of n and k of the map-reduce is:

$$S(k,i) = \begin{cases} 2.S(k, \frac{i}{2}) + kW \log i \\ \mathcal{A}(k,i) & \text{if } i < \frac{n}{s} \end{cases}$$

Assuming s constant and $H = W = \sqrt{n}$ this equation solves to:

$$S(k,n) = s.\mathcal{A}(k, \frac{n}{s}) + O(k.\sqrt{n} \log n \log s) = O(\mathcal{A}(k,n)) + O(k\sqrt{n} \log n)$$

When there is as much splits as rows, s in now dependent of n, it leads to Matas et al. [6] algorithm whose complexity is:

$$S(k,n) = H.W + O(k.W \log n \log H) \simeq O(n) + O(k\sqrt{n}(\log n)^2)$$

Contrary to what they claim, when values are small integers the complexity stays linear and is not dominated by merging operations. Finally, canonization and S reconstruction has a linear time complexity (`CanonizeRec` is called only once for each point) and only uses an image of n elements to track points already processed.

Table 1: Comparison of time complexity of many max-tree algorithms. n is the number of pixels and k the number of gray levels.

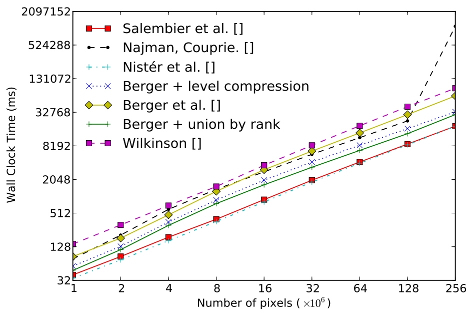
	Time Complexity		
Algorithm	Small int	Large int	Generic V
Berger [2]	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Berger + rank	$O(n.\alpha(n))$	$O(n \log \log n)$	$O(n \log n)$
Najman and Couprie [10]	$O(n.\alpha(n))$	$O(n \log \log n)$	$O(n \log n)$
Salembier et al. [13]	$O(n.k)$	$O(n.k) \simeq O(n^2)$	N/A
Nistér and Stewénius [11]	$O(n.k)$	$O(n.k) \simeq O(n^2)$	N/A
Wilkinson [20]	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Salembier non-recursive	$O(n.k)$	$O(n \log \log n)$	$O(n \log n)$
Menotti et al. [8] (1D)	$O(n)$	$O(n)$	$O(n)$
Map-reduce	$O(A(k,n))$	$O(A(k,n))+$ $O(k\sqrt{n} \log n)$	$O(A(k,n))+$ $O(k\sqrt{n} \log n)$
Matas et al. [6]	$O(n)$	$O(n) + O(k\sqrt{n}(\log n)^2)$	-

4 Experiments

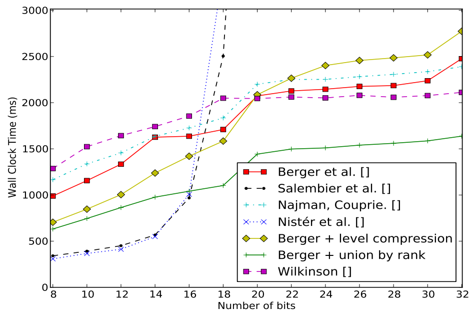
Benchmark were performed on an Intel Core i7 (4 physical cores, 8 logical cores). Program have been compiled with gcc 4.7, optimization flags on (`--O3 --march=native`). Tests have been conducted on a 6MB 8-bit image. Image has been resized by cropping or tiling the original image and over-quantization has

Table 2: Comparison of space requirements of many max-tree algorithms. n is the number of pixels and k the number of gray levels.

	Auxiliary space requirements		
Algorithm	Small int	Large int	Generic V
Berger et al. [2]	$n + k + stack$	$2n + stack$	$n + stack$
Berger + rank	$3n + k + stack$	$4.n + stack$	$3n + stack$
Najman and Couprie [10]	$5n + k + stack$	$6n + stack$	$5n + stack$
Salembier et al. [13]	$3k + n + stack$	$2k + n + stack$	N/A
Nistér and Stewénius [11]	$2k + 2n$	$2k + 2n$	N/A
Wilkinson [20]	$3n$	$3n$	$3n$
Salembier non-recursive	$2k + 2n$	$3n$	$3n$
Menotti et al. [8] (1D)	k	n	n
Map-reduce	$\ldots + n$	$\ldots + n$	$\ldots + n$



(a)



(b)

Fig. 4: (a) Comparison of algorithms on a 8-bit image as a function of size; (b) Comparison of algorithms on a 6 Mega-pixels image as a function of quantization.

been performed by shifting the eight bits left and generating missing lower bits at random. Figure 4 evaluates performance of sequential algorithms w.r.t to image size and quantization. A first remark, we notice that all algorithms are linear in practice. On natural image, the upper bound $n \log n$ complexity of Wilkinson [20] and Berger et al. [2] algorithms is not reached. Let start with union-find based algorithms. Berger et al. [2] and Najman and Couprie [10] have quite the same running time (±6% on average), however performances of Najman and Couprie [10] algorithm drops significantly at 256 Mega-pixels. Indeed, at that size each auxiliary array/image requires 1 GB memory space, thus Najman and Couprie [10] that uses much memory exceeds the 6 GB RAM limit and needs to swap with the hard drive. Our implementation of union-by-rank uses less memory and is on average 42% faster than Najman and Couprie [10]. Level compression is an efficient optimization that provides 35% speed up on average on Berger et al. [2]. However, this optimization is only reliable on low quantized data, figure 4b shows that it is relevant until 18 bits. Since it operates on flat-zones, when quan-

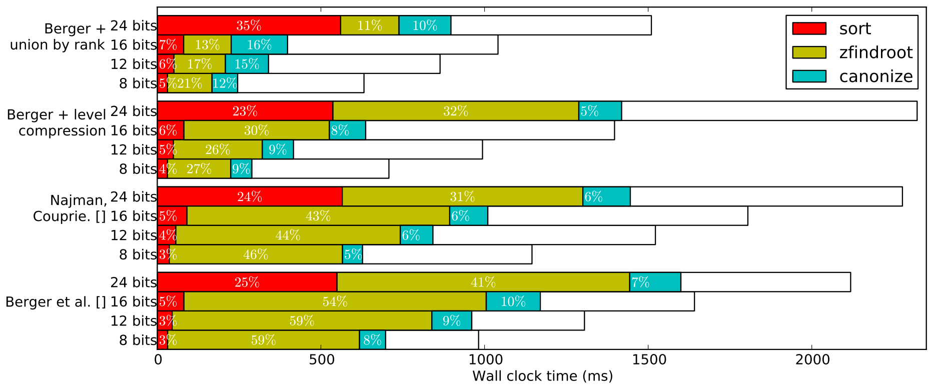
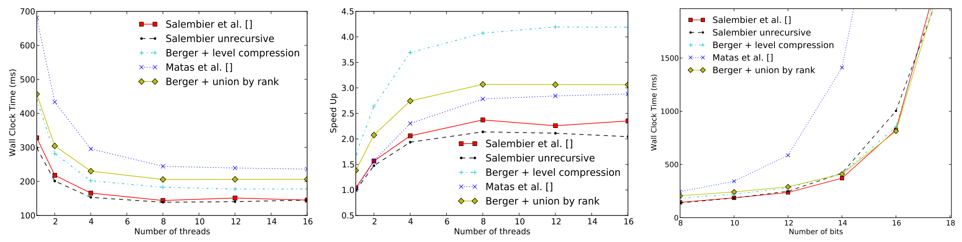


Fig. 5: Time spent in each part of the sequential version of union-find based algorithms.

tization gets higher flat-zones are less probable, thus time saved in FINDROOT to find canonical elements does not balance extra tests overheads (see Figure 5. Union-find is not affected by the quantization but sorting does, counting sort and radix sort complexities are respectively linear and logarithmic with the number of bits. The break in union-find curves between 18 and 20 bits stands for the switch from counting to radix sort. Flooding-based algorithms using hierarchical queues outperform our union-find by rank on low quantized image by 41% on average. As expected Salembier et al. [13] and Nistér and Stewénius [11] (which is the exact non-recursive version of the first one) closely match. However, the exponential cost of hierarchical queues w.r.t the number of bits is evident on figure 4b. By using a standard heap instead of hierarchical queues, Wilkinson [20] does scale well with the number of bits and outperform every algorithms except our implementation of union-by-rank. In [20], the algorithm is supposed to match Salembier et al. [13]'s method for low quantized images, but in our experiment it stays 4 times slower. As a consequence:

– since Najman and Couprie [10]'s algorithm is always outperformed by our implementation of union-find by rank, it will not be tested any further.
– [11] and [20] are merged in our single implementation (called *Non-recursive Salembier* below) that will use hierarchical queues for heap when quantization is below 18 and switches to a standard heap implementation otherwise.
– the algorithm *Berger + level compression* will enable level compression only when quantization is below 18 bits.

Figure 6 shows results of the map-reduce idiom applied on many algorithms and their parallelism. As a first result, we can see best performance are generally achieve with 8 threads that is when the number of thread matches the number of (logical) cores. However, since there are actually only 4 physical cores, we can expect a ×4 maximum speed up. Some algorithms take more benefits from map-reduce than others. Union-find based are algorithms are particularly well-



(a) (b) (c)

Fig. 6: (a,b) Comparison of parallel algorithms on a 6 Mega-pixels 8-bits image as a function of number of threads; (a) Wall clock time; (b) Speed up w.r.t the sequential version; (c) Comparison of parallel algorithms on a 6 Mega-pixels image as a function of quantization.

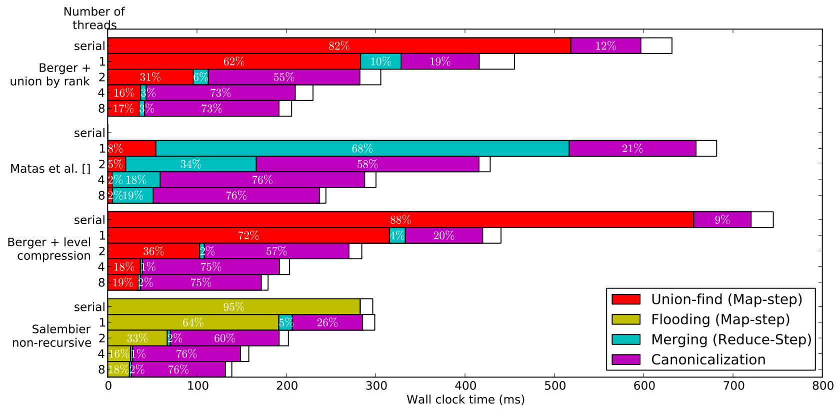


Fig. 7: Time spent in each part of parallel versions of algorithms.

suited for parallelism, union-find with level compression achieves the best speed up (×4.2) at 12 threads and union-find by rank a ×3.1 speed up with 8 threads. More surprising, map-reduce pattern achieves significant speed up even when a single thread is used (respectively ×1.7 and ×1.4 for union-find with level compression and union-find by rank). This result that used to surprise Wilkinson et al. [19] as well is explained by a better cache coherence when working on sub-domains that balances tree merges overhead. On the other hand, flooding algorithms do not scale as well because they are limited by the canonization and S reconstruction post-process (that is going to happen also for union-find algorithms on architecture with more cores). In [19] and [6], they obtain a speed up almost linear with the number of threads because only a *parent* image is built. If we remove canonization and S construction step, we also get those speed ups.(). Figure 6c shows the exponential complexity of merging trees as number

FiXme Fatal: Ajouter les diagrammes de rep des algo en steps of bits increases that makes parallel algorithms unsuitable for high quantized data. In the light of the previous analysis, Figure 8 provides guidelines to choose the right max-tree algorithm w.r.t to image types and architectures.

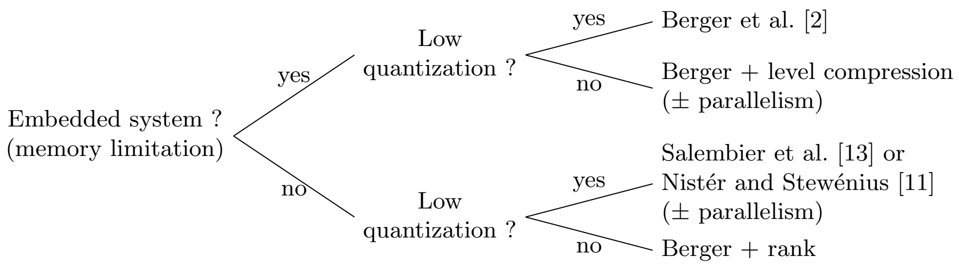


Fig. 8: Decision tree to choose the right max-tree algorithm

5 Conclusion

In this paper, we have tried to lead a fair comparison of max-tree algorithms in a unique framework. We have highlighted the fact that there is no best algorithm that supersedes all the others in every cases and eventually we have given a decision tree to choose the right algorithm w.r.t to data and hardware. We have proposed a max-tree algorithm using union-by-rank that outperforms the existing one from [10]. Furthermore, we have proposed a second one that uses level compression for systems with strict memory constraints. As further work, we shall include image contents as a new parameter of comparison, for instance images with large flat zones (e.g. cartoons) or images having strongly non-uniform distribution of gray levels. A code intensively tested used for these benachmarks is available on the Internet at `http://www.lrde.epita.fr/cgi-bin/twiki/view/Olena/maxtree`.

References

[1] Andersson, A., Hagerup, T., Nilsson, S., Raman, R.: Sorting in linear time? In: Proc. of the Annual ACM symposium on Theory of computing. pp. 427–436 (1995)

[2] Berger, C., Géraud, T., Levillain, R., Widynski, N., Baillard, A., Bertin, E.: Effective component tree computation with application to pattern recognition in astronomical imaging. In: Proc. of ICIP. vol. 4, pp. IV–41 (2007)

[3] Hesselink, W.: Salembier's min-tree algorithm turned into breadth first search. Information processing letters 88(5), 225–229 (2003)

[4] Jones, R.: Connected filtering and segmentation using component trees. Computer Vision and Image Understanding 75(3), 215–228 (1999)

[5] Matas, J., Chum, O., Urban, M., Pajdla, T.: Robust wide-baseline stereo from maximally stable extremal regions. IVC 22(10), 761–767 (2004)

[6] Matas, P., Dokladalova, E., Akil, M., Grandpierre, T., Najman, L., Poupa, M., Georgiev, V.: Parallel algorithm for concurrent computation of connected component tree. In: Advanced Concepts for Intelligent Vis. Sys. pp. 230–241 (2008)

[7] Meijster, A., Westenberg, M., Wilkinson, M.: Interactive shape preserving filtering and visualization of volumetric data. In: Proc. of ISIP. pp. 640–643 (2002)

[8] Menotti, D., Najman, L., de Albuquerque Araújo, A.: 1d component tree in linear time and space and its application to gray-level image multithresholding. In: Proc. of ISMM. pp. 437–448 (2007)

[9] Monasse, P., Guichard, F.: Fast computation of a contrast-invariant image representation. IEEE Trans. on Ima. Proc. 9(5), 860–872 (2000)

[10] Najman, L., Couprie, M.: Building the component tree in quasi-linear time. IEEE Trans. on Ima. Proc. 15(11), 3531–3539 (2006)

[11] Nistér, D., Stewénius, H.: Linear time maximally stable extremal regions. In: Proc. of ECCV. pp. 183–196 (2008)

[12] Perret, B., Lefevre, S., Collet, C., Slezak, É.: Connected component trees for multivariate image processing and applications in astronomy. In: Proc. of ICPR. pp. 4089–4092 (2010)

[13] Salembier, P., Oliveras, A., Garrido, L.: Antiextensive connected operators for image and sequence processing. IEEE Trans. on Ima. Proc. 7(4), 555–570 (1998)

[14] Salembier, P., Serra, J.: Flat zones filtering, connected operators, and filters by reconstruction. IEEE Trans. on Ima. Proc. 4(8), 1153–1160 (1995)

[15] Tarjan, R.: Efficiency of a good but not linear set union algorithm. JACM 22(2), 215–225 (1975)

[16] Urbach, E., Wilkinson, M.: Shape-only granulometries and grey-scale shape filters. In: Proc. of ISMM. vol. 2002, pp. 305–314 (2002)

[17] Vincent, L.: Grayscale area openings and closings, their efficient implementation and applications. In: Proc. of EURASIP Workshop on Mathematical Morphology and its Applications to Signal Processing. pp. 22–27 (1993)

[18] Wilkinson, M., Westenberg, M.: Shape preserving filament enhancement filtering. In: Proc. of MICCAI. pp. 770–777 (2001)

[19] Wilkinson, M., Gao, H., Hesselink, W., Jonker, J., Meijster, A.: Concurrent computation of attribute filters on shared memory parallel machines. PAMI 30(10), 1800–1813 (2008)

[20] Wilkinson, M.H.F.: A fast component-tree algorithm for high dynamic-range images and second generation connectivity. In: Proc. of ICIP. pp. 1021–1024 (2011)

[21] Xu, Y., Géraud, T., Najman, L., et al.: Morphological filtering in shape spaces: Applications using tree-based image representations. In: Proc. of ICPR. pp. 1–4 (2012)