A Root-to-Leaf Algorithm Computing the Tree of Shapes of an Image

Pascal Monasse

A Root-to-Leaf Algorithm Computing the Tree of Shapes of an Image

Pascal Monasse[0000−0001−9167−7882]

LIGM (UMR 8049), École des Ponts, UPE, Champs-sur-Marne, France

Abstract. We propose an algorithm computing the tree of shapes of an image, a unified variation of the component trees, proceeding from the root to the leaf shapes in a recursive fashion. It proceeds differently from existing algorithms that start from leaves, which are regional extrema of intensity, and build the intermediate shapes up to the root, which is the whole image. The advantage of the proposed method is a simpler, clearer, and more concise implementation, together with a more favorable running time on natural images. For integer-valued images, the complexity is proportional to the total variation, which is the memory size of the output tree, which makes the algorithm optimal.

Keywords: Tree of shapes · Component trees · Level sets.

1 Introduction

1.1 The Tree of Shapes

Extremal regions of an image are connected regions of an image where the intensity is above or below a certain gray level. These generalize the shapes of the classical mathematical morphology of binary images to gray-scale. Indeed, this just amounts to binarize the gray level image at a certain level and consider the resulting shapes. As there is not a single threshold where interesting shapes occur, all possible thresholds should be applied. For example, for an 8-bit image, all integer values from 0 to 255 can be used as thresholds. It is clear that when the threshold increases, minimal regions (those below the threshold) increase with respect to set inclusion, while maximal regions decrease. From these simple observations, two trees can be built, the minimal and the maximal trees, called the component trees. In each tree, a shape is an ancestor of another if the first contains the second. The root is the full set of pixels, obtained at threshold 255 in the min-tree and 0 in the max-tree. These observations are at the basis of efficient algorithms to compute the component trees, either bottom-up, i.e., from the leaves to the root [20,2], or top-down, from the root to the leaves [22,21]. While some of these algorithms are very efficient for 8-bit images, others beat the former on higher bit depths. A full comparison is available in the literature [4].

Naturally, not all extremal regions are significant or are the projection of a single 3D object in the image. However, a simple criterion, like a high contrast, can be enough to recover some important shapes that may be used as features

in image registration or disparity estimation. This is the principle at the basis of maximally stable extremal regions (MSER) [14], which yield point correspondences between images of the same scene in the same manner as the similarity invariant feature transform (SIFT) [13] and its variants. A more recent alternative to MSER is shapes just before their merging in a component tree, so-called tree-based Morse regions [26]. Some of these methods are compared in a famous study [16].

The need for extraction of both component trees can be lifted by using the tree of shapes [18]. The shapes involved in this construction are built from the connected components of extremal regions. The internal holes of the latter, that is all bounded connected components of their complement except one, the exterior, are filled, yielding the shapes. It happens that shapes, whether issued from minimal or maximal extremal regions, can still be ordered in an inclusion tree [1]. This unique structure is well suited for contrast-invariant filtering [19,6], self-dual filtering [12,10], segmentation [24,25], or image registration [17,9].

1.2 Related Work

The computation of the tree of shapes can be done by merging the component trees [5]. The advantage is that the algorithm works in any dimension, but it is not particularly efficient for 2D images. The standard algorithm, the fast level set transform (FLST) [7], works in a bottom-up fashion, starting from the leaves and leveling the image after each extraction of shape [15]. Interpreting the image as a continuous bilinear interpolated surface yields the tree of bilinear level lines, which can be efficiently computed based on level lines [7,8], which is akin to the proposed algorithm. The closer to the proposed algorithm is top-down [23], from root to leaves, but only applicable to hexagonal connectivity, while ours works in the standard 4- and 8-connectivity. Finally, a different definition of shapes based on multi-valued images [11] has the potential to be computed very efficiently, but no public implementation seems available.

1.3 Background and Notations

We consider discrete images I defined on pixels. Each pixel $p \in \{0,\dots,w-1\} \times \{0,\dots,h-1\}$ gets a value $I(p) \in \mathbb{R}$. We consider the extremal regions, or level sets: $\{p : I(p) \leq \lambda\}$ and $\{p : I(p) \geq \lambda\}$. The 4-connected components of the former are called inferior components and the 8-connected components of the latter are called superior components. The asymmetry is necessary here to get an inclusion tree later on. Assuming a component C not touching the image boundary $\{0,w-1\} \times \{0,\dots,h-1\} \cup \{0,\dots,w-1\} \times \{0,h-1\}$, we can fill all connected components of its complement (8-connected if C is an inferior component, 4-connected otherwise) except the one containing the image boundary, yielding a shape S. The shapes built from all such components, together with the set of all pixels R, has an inclusion tree structure: A shape S is an ancestor of another S' iff $S' \subset S$. If S and S' are not nested like here, then $S \cap S' = \emptyset$.

Since any pixel p is inside a shape (at least R), we can consider all shapes containing p, which are nested since they intersect; the smallest of them is noted $S[p]$. Pixels $p_1, \ldots, p_k$ such that $S = S[p_1] = \cdots = S[p_k]$ are called the private pixels of S. In that case, we must have $I[p_1] = \cdots = I[p_k]$, called the gray level g of S. A pixel p in S having a 4-neighbor (if S is inferior) or 8-neighbor (if S is superior) q outside S is said to be at the boundary of S, while q is said to be an external neighbor of S. It seems that all private pixels of S are at its boundary or connected to such a pixel inside the iso-level $I_g := \{p : I(p) = g\}$. Actually, this is not all, since all pixels at the external boundary of a *child* shape of S in the tree having gray level g and their connected components in I_g are also private pixels. This is illustrated Figure 1. The private pixels of shape B are two components of iso-level at the boundary, together with another component *not* at the boundary, but at the immediate exterior of C and D, children of B.

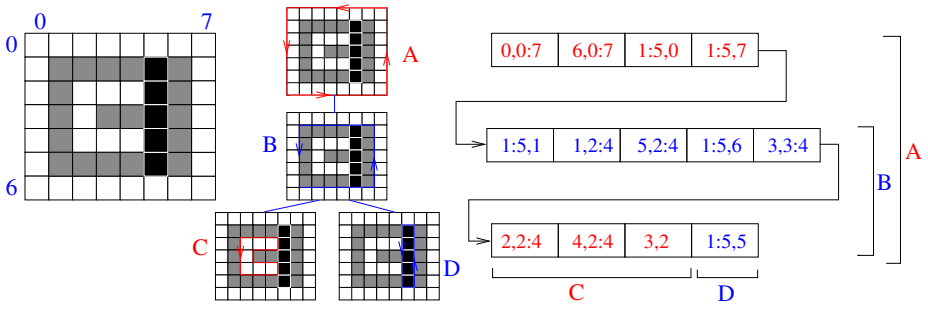


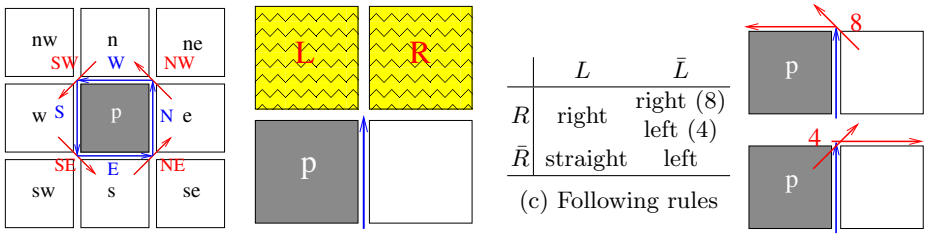
Fig. 1. Tree of shapes of a 7×8 image. The boundaries (level lines) of superior shapes are in red and of inferior shapes in blue. The root is A, its child is B, whose children are C and D. On the right, the arrangement of pixels according to their smallest shape (Matlab range notation).

2 Following a Level Line

The key element of the algorithm is that it is based on level lines rather than level sets. A shape is the "interior region" delimited by a level line and the level line is the boundary of the shape. Each level line is trodden twice by the algorithm: the first time to extract the level line itself and find the gray level of the shape, along with one private pixel; the second time to find possible additional private pixels of the parent shape that are not connected to its boundary.

The level line is a sequence of consecutive edgels. An edgel ("edge element") can be represented as the common boundary between two adjacent pixels, that is pixels for which one or two coordinates, x and y, differ by one unit. In order to ensure that each pixel is involved in exactly 8 edgels and to avoid exceptions for pixels at the boundary of the image, we represent rather an edgel by one pixel

and one cardinal direction: east (`E`), north (`N`), west (`W`) or south (`S`), and the diagonal directions. An edgel is thus *oriented*, and the pixel is called its interior pixel, while its exterior pixel, if it exists, is the pixel adjacent to the interior one across the direction, see Figure 2(a). Consecutive edgels either share a common interior pixel (we say the level line "turns left") or a common exterior pixel (the level line "turns right"), or have their interior pixels adjacent along their identical directions (the level line "goes straight").



(a) Edgels (b) Direction choice

	L	$\bar{L}$
R	right	right (8) left (4)
$\bar{R}$	straight	left

(c) Following rules

(d) Turns

Fig. 2. Edgels and level lines following. (a) The edgels associated to interior pixel p are for example $(p, s) = (p, E)$ and $(p, ne) = (p, NW)$, the latter being diagonal. (b) When following a level line (here at edgel (p, N)), the new edgel depends on whether L and R are within the shape. (c) Rules for next edgel, depending on whether L and R are above the threshold or not ($\bar{L}$ and $\bar{R}$). (d) Turns left and right. The diagonal direction is followed iff the connexity matches the displayed number.

To be a level line of the image I, a sequence of consecutive edgels $L = (e_0, \dots, e_{n-1}, e_n = e_0)$, with $e_i \neq e_j$ for $0 \leq i < j < n$, must satisfy

$$g := \max_{0 \leq i < n} I(\mathrm{Int}(e_i)) < \min_{0 \leq i < n} I(\mathrm{Ext}(e_i)) \text{ or} \tag{1}$$

$$g := \min_{0 \leq i < n} I(\mathrm{Int}(e_i)) > \max_{0 \leq i < n} I(\mathrm{Ext}(e_i)). \tag{2}$$

In the first case, we say that L is an inferior level line, which is the boundary of an inferior shape, while in the second case L is a superior level line, boundary of a superior shape. In any case, the gray level g is called the level of L and of the shape whose boundary is L. In the above formulas, intensities of nonexistent exterior pixels are replaced by $-\infty$. If no exterior pixel exists in the sequence, that is, L follows the boundary of I, the right-hand side becomes $-\infty$ and we are in the second alternative, a superior level line that is the boundary of the root shape. In general, any edgel with no exterior pixel is of superior type[1]. Their order indicates the type of the level line.

As soon as we have an edgel e with $I(\mathrm{Int}(e)) \neq I(\mathrm{Ext}(e))$, e is in a level line of I. In our algorithm, we need to find the largest shape, in the sense of set

[1] This choice is arbitrary, we could have chosen $+\infty$ and the root shape would have been of inferior type

inclusion, whose boundary L includes e, that is, the one whose gray level g is as close as possible to $I(\text{Ext}(e))$, which may be different from $I(\text{Int}(e))$. Starting from e, we need to find the following one in the sequence $\mathcal{L}(e)$ we are building. We iterate this until we reach again e. For this, the procedure needs to know which direction to follow (left turn, right turn, or straight ahead). This depends on the gray levels of two pixels, L and R in Figure 2(b). The rules for finding the next edgel are in Figure 2(c) according to whether L (resp. R) is in the shape or not, the latter case being noted $\bar{L}$ (resp. $\bar{R}$). The rule indicates which direction to take: straight means continue to the direction of the arrow, that is, the interior pixel becomes L and the direction does not change. The indications "left" and "right" indicate a turning direction. In a left turn, the interior pixel remains the same, but not in a right turn. The complex case is $\bar{L} \wedge R$, which is a saddle point. The turn to take depends on the connexity: left for 4-connexity and right for 8-connexity. Performing a turn is explained by Figure 2(d). A left turn is performed in two steps in 8-connexity, first with a diagonal direction (current direction N would be followed by NW, then W in the illustration), and also for a right turn in 4-connexity (NE then E). The procedure to follow a level line is summarized in Algorithm 1. According to Figure 2(d), left and right turns can actually generate two edgels depending on connexity. In such case, the first edgel appended is diagonal, and the following edgel finishes the turn.

Data: Edgel e_0, with $I(\text{Int}(e_0)) \neq I(\text{Ext}(e_0))$
Result: $\mathcal{L}(e_0) = (e_0, \ldots, e_{n-1}, e_n = e_0)$: largest level line through e_0
$\lambda \leftarrow I(\text{Ext}(e_0))$
repeat
  $l \leftarrow \text{Int}(\texttt{go\_straight}(e_i))$, $r \leftarrow \text{Ext}(\texttt{go\_straight}(e_i))$
  $L \leftarrow \text{sign}(I(l) - \lambda) = \text{sign}(I(\text{Int}(e_0)) - \lambda)$
  $R \leftarrow \text{sign}(I(r) - \lambda) = \text{sign}(I(\text{Int}(e_0)) - \lambda)$
  if $L \wedge R$ **then** $e_{i+1} \leftarrow \texttt{turn\_right}(e_i)$
  if $L \wedge \bar{R}$ **then** $e_{i+1} \leftarrow \texttt{go\_straight}(e_i)$
  if $\bar{L} \wedge \bar{R}$ **then** $e_{i+1} \leftarrow \texttt{turn\_left}(e_i)$
  if $\bar{L} \wedge R$ **then** `// Saddle point: turn depending on connexity`
    if $I(\text{Int}(e_0)) > \lambda$ **then** $e_{i+1} \leftarrow \texttt{turn\_right}(e_i)$
    else $e_{i+1} \leftarrow \texttt{turn\_left}(e_i)$
  $i \leftarrow i + 1$
until $e_i = e_0$

Algorithm 1: Follow a level line from an initial edgel

3 Top-Down Algorithm

3.1 Representation of the Tree

A shape is stored as a structure comprising its type (inferior or superior), its level g, its contour (array of positions) and an array of its pixels. Moreover, it contains

pointers to its parent shape, its "first" child, and its next sibling, if any. In that manner, all children of a shape form a list structure. This is all that is required for walking the tree in any way. Pixels are stored taking advantage of nesting: private pixels of a shape come first in the array, followed by arrays of pixels of its children. This recursive ordering allows to have a single array containing a permutation of all pixels of the image, and each shape has a pointer for its beginning and a pointer to its end inside this array (see Figure 1). Notice also that such an arrangement provides an $O(1)$ procedure to determine whether one shape is a descendant of another, by comparison of pointers.

The tree is represented as an array of shapes[2]. It stores also an index `S[p]`, giving for each pixel p the shape of which it is a private pixel.

3.2 Top-Down Recursive Extraction

The algorithm 2 starts from an edgel e whose internal and external pixels have different intensities. The procedure `create_tree` builds the shape S whose boundary is $\mathcal{L}(e)$, and recursively the tree rooted at S. In order to do that, the first loop of the algorithm follows $\mathcal{L}(e)$ and stores a single internal pixel p, which is a private pixel of S. Its gray level g is the closest to $I(\mathrm{Ext}(e))$ among all internal pixels of edges in $\mathcal{L}(e)$. This loop also reinitializes $S[.]$ to $\emptyset$ at each interior pixel. This is necessary since the call of `find_pp_children` from the parent P (procedure detailed below) has overwritten it with tag P. After this loop, the level g of S is stored and the registered private pixel p is put into a queue Q. The tag $S[p]$ is set to S, like all subsequent pixels pushed into Q.

Data: edgel e
Result: Tree rooted at largest shape S whose level line L goes through e
$\lambda \leftarrow I(\mathrm{Ext}(e))$ `// Level of parent`
$p \leftarrow \mathrm{Int}(e)$
for $e_i \in \mathcal{L}(e)$ **do** `// Line following, Algorithm 1`
  $S[\mathrm{Int}(e_i)] \leftarrow \emptyset$
  if $|I(\mathrm{Int}(e_i)) - \lambda| < |I(p) - \lambda|$ **then**
    $p \leftarrow \mathrm{Int}(e_i)$
$S.\mathrm{type} \leftarrow \begin{cases} \inf & \text{if } I(p) < \lambda \\ \sup & \text{if } I(p) > \lambda \end{cases}$
$S.g \leftarrow I(p)$, $S[p] \leftarrow S$
$C \leftarrow$ `find_pp_children`(S, p) `// C is an array of edgels`
for $e_i \in C$ **do**
  $S' \leftarrow$ `create_tree`(e_i) `// Recursive call for children`
  Insert S' as child of S

Algorithm 2: `create_tree`, main routine

[2] The first shape in the array is the root, corresponding to the full image, and parents have a position before their children. This is a consequence of the top-down nature of the algorithm.

All pixels that will transit in Q will be the private pixels of S. The second step (Algorithm 3) dequeues the waiting pixel p from Q and examines its neighbors (4- or 8-neighbors depending on the type of S). For each one q not already explored ($S[p] = \emptyset$), we have two possibilities: if $I[p] = I[q]$, q is also a private pixel, it is marked as explored ($S[q] = S$) and inserted in Q. Otherwise, the edgel $e = (q, p)$ is on a level line $\mathcal{L}(e)$ bounding a shape that is a child of S. This edgel is put in an array C for later treatment. The level line $\mathcal{L}(e)$ is followed, with a twofold goal: mark internal pixels r as explored ($S[r] = S$) so as to prevent a re-exploration from a different edgel; if an exterior pixel r is unmarked and $I(p) = g$, mark r and enqueue it in Q, since it is a private pixel.

```
Data: Shape S, one private pixel p_0
Result: Find all private pixels of S; Return array C of edgels, one per child
        level line.
Q ← p_0                                                  // Push p_0 in a queue
while Q ≠ ∅ do
    Q → p                                          // Pop pixel from Q, store in p
    Add p as private pixel of S
    for q ∼ p and S[q] = ∅ do                      // Unexplored neighbors of p
        S[q] ← S
        if I(q) = I(p) then
            Q ← q                                          // Push q, private pixel
        else
            e = (q, p)                            // Edgel with q as interior pixel
            C ← e
            for e_i ∈ L(e) do
                S[Int(e_i)] ← S
                if S[Ext(e_i)] = ∅ and I(Ext(e_i)) = S.g then
                    Q ← Ext(e_i)                          // This is a private pixel
                    S[Ext(e_i)] ← S
```

Algorithm 3: `find_pp_children`, find private pixels of S and one edge per child level line

Each edgel in C is at the boundary of a different child of S. At this point, all internal and all external pixels of an edgel along the boundary of S is marked by $S[p] = S$. Each one provokes a recursive call to `create_tree`, whose first step puts back internal pixels along the boundary to $\emptyset$, as seen above.

3.3 Complexity

At any edgel e, there are at most $|I(\mathrm{Int}(e)) - I(\mathrm{Ext}(e))|$ level lines going through e if I takes only integer values. Each level line is followed twice, so that the complexity of the algorithm is

$$O(\sum_{p \sim p'} |I(p) - I(p')|) = O(\mathrm{TV}(I)), \tag{3}$$

the (discrete) total variation of I. Notice that if level lines are stored with the shapes, this complexity is asymptotically optimal, since it is proportional to the output size. However, for large images storing the level lines can be too demanding, and our code makes this storage optional at compile time.

3.4 Comparison with the FLST

The reference algorithm for extraction of the tree of shapes is the FLST [7]. Its high level operation is summarized in Algorithm 4. Its procedure is bottom-up, that is, it starts from leaves and goes up to the root of the tree.

```
1  for pixel p do
2      if p local extremum then
3          while true do
4              Extract iso-level S = cc({I = I(p)}, p)
5              if S regional extremum without hole then
6                  Insert S as new shape in tree
7                  for q ∈ S do
8                      if S[q] = ∅ then
9                          S[q] ← S
10                     else
11                         I(q) ← I(p)
12                         Add highest ancestor of S[q] as child of S
13                 p ← neighbor of S of closest intensity to I(p)
14             else
15                 break while loop
```

Algorithm 4: High-level operation of the FLST.

It relies on the observation that a leaf of the tree of shapes is a regional extremum of the image with hole, and that a regional extremum contains a local extremum. Pixels are thus sequentially scanned, and when a local extremum p is met, the procedure tries to extract a shape and its ancestors:

1. The set S of pixels connected to p at the same level $I(p)$ are extracted by region growing;
2. If S is a regional extremum without hole then S is a new shape, all pixels q of S with no smallest associated yet are private pixels of S, while other pixels of S are set to level $I(p)$ and their largest ancestor yet (upper-most parent of $S[q]$ is set as a child of S.
3. p is moved to a neighbor of S of closest intensity to $I(p)$ and we continue to step 1, that is we try going up the tree. It can be noticed that the new S will be a superset of the current one, so the region growing needs not start from the single pixel p.

The bottleneck of the algorithm is the abortion of the `while` loop at line 15, whose goal is to walk up the tree (to the root), when the set S presents one or several internal holes: these holes are filled one by one at line 11, but only the last one is able to proceed up the tree and extract S as a shape.

The worst case scenario for the FLST is in the presence of a large uniform area with many holes, for example a checkerboard. Each black case of one pixel is a hole in the white shape, so that the complexity is $O(n^2)$ with n the number of pixels. By contrast, the proposed algorithm does not have any particular trouble with such a situation, yielding a complexity $O(n)$.

On the contrary, when a deep hierarchy of nested shapes is present, the proposed algorithm is quite slow, since a new level line has to be followed for each one. This worst situation can happen for high bit-depth image, typically when each pixel has its own single gray level. There are as many shapes as pixels, and the length of level lines could be also large. Another trouble is that the recursivity follows the tree depth and could result in a memory stack overflow. In this situation, the FLST has no difficulty at all, just walking up the tree shape by shape without break.

Roughly speaking, the proposed algorithm is advantageous for low bit-depth images and wide (many children) but shallow trees, while the FLST prefers narrow (few children) but deep trees. It happens that for 8-bit images, the former is more frequent than the latter. This is demonstrated by the experiments of the next section.

4 Experiments

Our implementation[3] is coded in `C++`. The core of the algorithm is about 250 lines of code. It is compiled with the GNU compiler `gcc` version 4.8 in optimized mode and run on an Intel Xeon CPU E5-2643 at 3.3 GHz. Experiments are performed on different crops of large Wikipedia images. Each crop is at the center of image. Run times of the proposed algorithm and of the classical FLST are in Figure 3.

On the *Church* image[4], a regular photograph, both the proposed algorithm and the classical FLST are performing well, with a significant advantage for the proposed algorithm beginning at about 10 Mpixels. Notice the time spent for our algorithm is roughly linear, so as the TV. For the *Meteo* image[5], a satellite image, our algorithm is quite fast even at high resolution, while the classical FLST seems to have a super-linear running time. Speed-ups of our algorithm are very significant here. Finally, for a simple *Cartoon* image[6], with a low TV, our algorithm performs very well (5 seconds at 20 Mpixels), while the classical FLST takes above 120 seconds at 5 Mpixels, and more than 500 seconds at 9 Mpixels. This is because such a situation is not favorable to the FLST, with some shapes having a high number of children. Each time a child is extracted, the FLST levels it to the gray level of the parent, until it realizes that more children are to be

[3] https://github.com/pmonasse/flst

[4] https://upload.wikimedia.org/wikipedia/commons/5/5e/12-04-06-senftenhuette-by-RalfR-08.jpg

[5] https://upload.wikimedia.org/wikipedia/commons/6/6b/02S_Dec_8_2011_0600Z.jpg

[6] https://upload.wikimedia.org/wikipedia/commons/6/6c/0-cynefin-ORIGINEEL.jpg

extracted. In such a situation, the path going up the tree has to stop until all children are extracted. The proposed algorithm has no such problem.

The proposed algorithm extracts the tree of shapes of *Church* (60 Mpixels) in 54 s, of *Meteo* (56 Mpixels) in 32 s, and of *Cartoon* (300 Mpixels) in 85 s. This shows that the algorithm, although single thread, is able to handle large images in reasonable time.

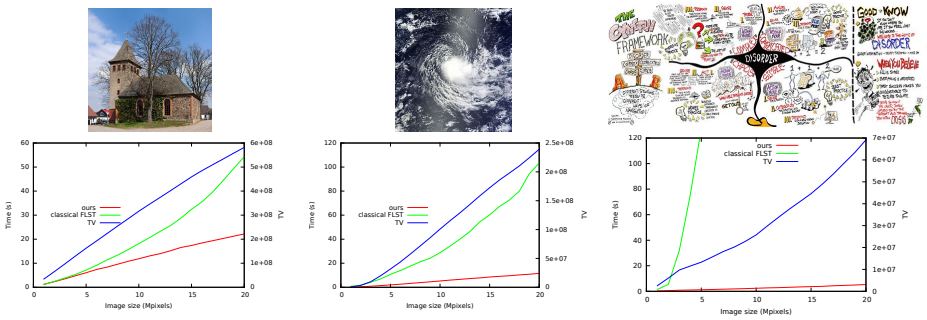


Fig. 3. Run-time with respect to image size.

5 Conclusion

We have presented an algorithm that computes the tree of shapes by starting from the root and proceeding downward to the leaves. Experiments show that it is more efficient on natural images, and the algorithm is even optimal on integer-valued images.

One minor drawback is that the current implementation assumes a priori a fixed gray level outside the image frame, whereas its prime concurrent, the FLST, adapts to the image contents. A possible solution to recover this feature is to compute two trees, one when the outside is below the minimal value in the image and the other where it is above the maximal value. However, the root-to-leaf extraction of the second tree can be stopped as soon as shapes do not meet the image boundary, because these shapes are common to both trees. It remains only to sort the shapes meeting the boundary from both trees to put them in the correct hierarchy [5].

Further experiments would be needed in order to compare with a modified version of the FLST that extracts the full tree in several passes with increasing maximal area [7], which alleviates the problem of wide trees. Another worthwile comparison is with a quasi-linear algorithm that transforms the extraction to a max-tree computation into a larger image [3].

A possible extension of the algorithm is to derive a parallel implementation for handling extra-large images. Indeed, since extracting a subtree rooted at some node of the tree has no side-effect on its exterior, i.e., the rest of the image,

different threads could handle the bifurcations in the tree (several children to common parent), each one building the subtree rooted at a sibling, without need for synchronization. This is in strong contrast to bottom-up algorithms where threads building up from different leaves would need to wait for each other when they have to merge at their common ancestor.

References

1. Ballester, C., Caselles, V., Monasse, P.: The tree of shapes of an image. ESAIM: COCV **9**, 1–18 (2003)
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: Image Processing, 2007. ICIP 2007. IEEE International Conference on. vol. 4, pp. IV–41. IEEE (2007)
3. Carlinet, E., Crozet, S., Géraud, T.: The tree of shapes turned into a max-tree: A simple and efficient linear algorithm. In: Proceedings of the IEEE International Conference on Image Processing (ICIP) (2018)
4. Carlinet, E., Géraud, T.: A comparative review of component tree computation algorithms. IEEE Transactions on Image Processing **23**(9), 3885–3895 (2014)
5. Caselles, V., Meinhardt, E., Monasse, P.: Constructing the tree of shapes of an image by fusion of the trees of connected components of upper and lower level sets. Positivity **12**(1), 55–73 (2008)
6. Caselles, V., Monasse, P.: Grain filters. Journal of Mathematical Imaging and Vision **17**(3), 249–270 (2002)
7. Caselles, V., Monasse, P.: Geometric description of images as topographic maps, Lecture Notes in Computer Science, vol. 1984. Springer (2009)
8. Ciomaga, A., Monasse, P., Morel, J.M.: The image curvature microscope: Accurate curvature computation at subpixel resolution. Image Processing On Line **7**, 197–217 (2017), https://doi.org/10.5201/ipol.2017.212
9. Dibos, F., Koepfler, G., Monasse, P.: Image alignment. In: Geometric Level Set Methods in Imaging, Vision, and Graphics, pp. 271–295. Springer (2003)
10. Dibos, F., Koepfler, G., Monasse, P.: Total Variation Minimization for Scalar/Vector Regularization, pp. 121–140. Springer New York, New York, NY (2003)
11. Géraud, T., Carlinet, E., Crozet, S., Najman, L.: A quasi-linear algorithm to compute the tree of shapes of nd images. In: International Symposium on Mathematical Morphology and Its Applications to Signal and Image Processing. pp. 98–110. Springer (2013)
12. Keshet, R.: Shape-tree semilattice. Journal of mathematical imaging and vision **22**(2-3), 309–331 (2005)
13. Lowe, D.G.: Object recognition from local scale-invariant features. In: Computer vision, 1999. The proceedings of the seventh IEEE international conference on. vol. 2, pp. 1150–1157. Ieee (1999)
14. Matas, J., Chum, O., Urban, M., Pajdla, T.: Robust wide-baseline stereo from maximally stable extremal regions. Image and vision computing **22**(10), 761–767 (2004)
15. Meyer, F., Maragos, P.: Nonlinear scale-space representation with morphological levelings. Journal of Visual Communication and Image Representation **11**(2), 245–265 (2000)

16. Mikolajczyk, K., Schmid, C.: A performance evaluation of local descriptors. IEEE transactions on pattern analysis and machine intelligence **27**(10), 1615–1630 (2005)
17. Monasse, P.: Contrast invariant registration of images. In: Acoustics, Speech, and Signal Processing, 1999. Proceedings., 1999 IEEE International Conference on. vol. 6, pp. 3221–3224. IEEE (1999)
18. Monasse, P., Guichard, F.: Fast computation of a contrast-invariant image representation. IEEE Transactions on Image Processing **9**(5), 860–872 (2000)
19. Monasse, P., Guichard, F.: Scale-space from a level lines tree. Journal of Visual Communication and Image Representation **11**(2), 224–236 (2000)
20. Najman, L., Couprie, M.: Building the component tree in quasi-linear time. IEEE Transactions on image processing **15**(11), 3531–3539 (2006)
21. Nistér, D., Stewénius, H.: Linear time maximally stable extremal regions. In: European Conference on Computer Vision. pp. 183–196. Springer (2008)
22. Salembier, P., Oliveras, A., Garrido, L.: Antiextensive connected operators for image and sequence processing. IEEE Transactions on Image Processing **7**(4), 555–570 (1998)
23. Song, Y.: A topdown algorithm for computation of level line trees. IEEE Transactions on Image Processing **16**(8), 2107–2116 (2007)
24. Xu, Y., Carlinet, E., Géraud, T., Najman, L.: Hierarchical segmentation using tree-based shape spaces. IEEE transactions on pattern analysis and machine intelligence **39**(3), 457–469 (2017)
25. Xu, Y., Géraud, T., Najman, L.: Context-based energy estimator: Application to object segmentation on the tree of shapes. In: Image Processing (ICIP), 2012 19th IEEE International Conference on. pp. 1577–1580. IEEE (2012)
26. Xu, Y., Monasse, P., Géraud, T., Najman, L.: Tree-based Morse regions: A topological approach to local feature detection. IEEE Transactions on Image Processing **23**(12), 5612–5625 (2014)