Chapter 3

Max-Tree

3.1 Definition

The Max-Tree is a structured representation of the connected components of the level sets an image is made of. This representation was developed to be able to deal with classical anti-extensive connected operators, as well as new ones, in an efficient manner. Due to this initial strategy the Max-Trees is only able to represent binary or gray-level images, but not multicomponent images.

In order to construct the tree, the image is considered to be a 3D relief, as shown in Fig. 3.1. The nodes of the tree represent the connected components of the upper level sets for all possible gray-level values. The leafs of the tree correspond to the regional maxima of the image. The links between the nodes describe the inclusion relationship of the binary connected components.

We want now to formally define the region of support of each node. Let $X_h(f)$ denote the upper level set resulting from thresholding the function f at gray value h. The connected components of $X_h(f)$ are labelled with a labelling algorithm, using either classical 4 or 8 neighboring connectivity. Let $UpCC_h^k$, $k \geq 1$, denote the k'th connected component of

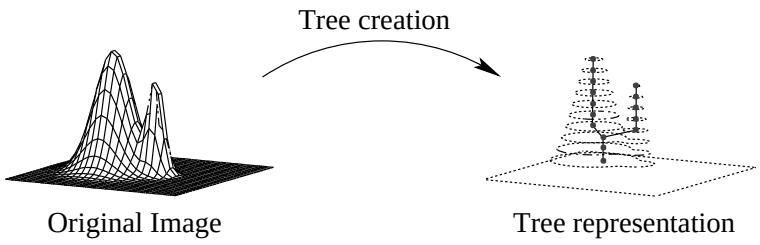


Figure 3.1: Constructing the Max-Tree by considering the image to be a 3D relief.

$X_h(f)$. In the Max-Tree representation each node N_j is associated to one such connected components[1].

In order to establish the parent relationship between nodes, the connected components associated to $X_h(f)$ and $X_{h+1}(f)$ are studied. The node N_{j_2} is a child of the node N_{j_1} if its respective associated connected components $UpCC^i_{h+1}$ and $UpCC^k_h$ satisfy:

$$N_{j_1} \rightarrow N_{j_2} \quad \text{iff} \quad UpCC^i_{h+1} \subseteq UpCC^k_h$$

That is, the links between nodes of the tree representation denote inclusion relationship between connected components at different gray-levels. Note that a node may have several children, and therefore the tree structure may be arbitrary.

The Max-Tree is therefore a hierarchical representation of the upper level sets. It should be noted that the set of upper level sets do not form partition hierarchy but a decomposition of the image into a set of regions.

Sec. 2.1 studies the relationship between flat zones and the level sets. In particular, the upper level set of gray-level h is composed by the set of flat zones of gray-level h and the upper level set of gray-level $h+1$. Thus, in order to avoid redundancy in the tree representation, each node N_j only needs to hold the flat zones of gray-level h included in its associated connected component $UpCC^k_h$. Using this approach, the connected component $UpCC^k_h$ associated to a node N_j can be extracted by taking the union of the set of flat zones associated to its descendants. With the proposed strategy, we may say that therefore the Max-Tree can also be viewed as a structured representation of the partition of flat zones an image is made of. Moreover, the efficient algorithm to construct the tree, described in Sec. 3.4, is based on the this approach to construct the tree.

In Fig. 3.2 an example of Max-Tree construction is shown. The original image is composed of seven flat zones identified by a letter $\{A, B, C, D, E, F, G\}$. The number following each letter defines the gray level value of the flat zone. In our example, the gray-level values range from 0 to 2. In the first step, the threshold h is fixed to the gray-level value 0. The image is binarized: all pixels at level $h = 0$ (pixels of region A) are assigned to the root node of the tree $UpCC^1_0 = \{A\}$. Furthermore, the pixels of gray-level value strictly higher than $h = 0$ form two connected components $UpTCC^1_1 = \{G\}$ and $UpTCC^2_1 = \{B, C, D, E, F\}$ that are temporally assigned to nodes $UpCC^1_1$ and $UpCC^2_1$ respectively. This creates the first tree (for gray-levels $[0, 1]$). In a second step, the threshold is increased by one: $h = 1$. Each node $UpTCC^k_{h=1}$ is processed as the original image. Consider, for instance, the node $UpTCC^2_1 = \{B, C, D, E, F\}$: all its pixels at level $h = 1$ remain unchanged and create the final node $UpCC^2_1$. However, pixels of gray-level values strictly higher than h (here $\{E, C\}$) create two different connected components and are moved to two temporary child nodes $UpCC^2_2 = \{C\}$ and $UpCC^3_2 = \{E\}$.

[1]To simplify, when referring to a node in the Max-Tree we may say "the node $UpCC^k_h$" instead of "the node N_j whose associated connected component is $UpCC^k_h$".

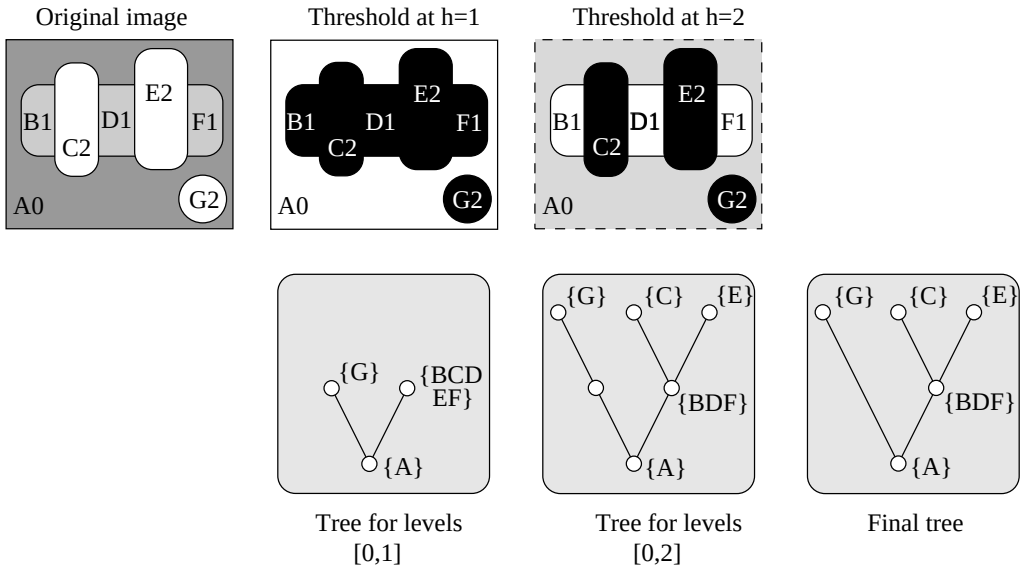


Figure 3.2: Max-Tree representation creation example.

The complete tree construction is done by iterating this process for all nodes k at level h and for all possible thresholds h (from 0 to the highest gray-level value). The algorithm can be summarized saying that, at each temporary node $UpTCC_h^k$, a "local" background is defined by keeping all pixels of gray-level value equal to h (the "local" background itself may not be connected) and that the various connected components formed by the pixels of gray-level value higher than h create the child nodes of the tree. In this procedure, some nodes may become empty. Therefore, at the end of the tree construction, the empty nodes are removed since they do not contribute any relevant information to the tree structure.

Note that the description of the previous technique does not necessarily correspond to the actual implementation of the tree construction. The tree should be constructed in such a way that each pixel is analyzed the least number of possible times. Sec. 3.4 is thus devoted to the efficient construction of the tree.

Fig. 3.3 (resp. Fig. 3.4) shows an example of Max-Tree representation. The original image is made up of 150 (resp. 100) flat zones. The associated Max-Tree is depicted, and on some nodes the associated connected components are shown.

The final tree is called a Max-Tree in the sense that it is a structured representation oriented towards the maxima of the image. In fact, the leaves of the tree represent the regional maxima of the image, whereas the rest of the nodes are ordered along the tree branches according to the gray-level value of the corresponding flat zones. Finally, the root node corresponds to the lowest gray-level value of the image.

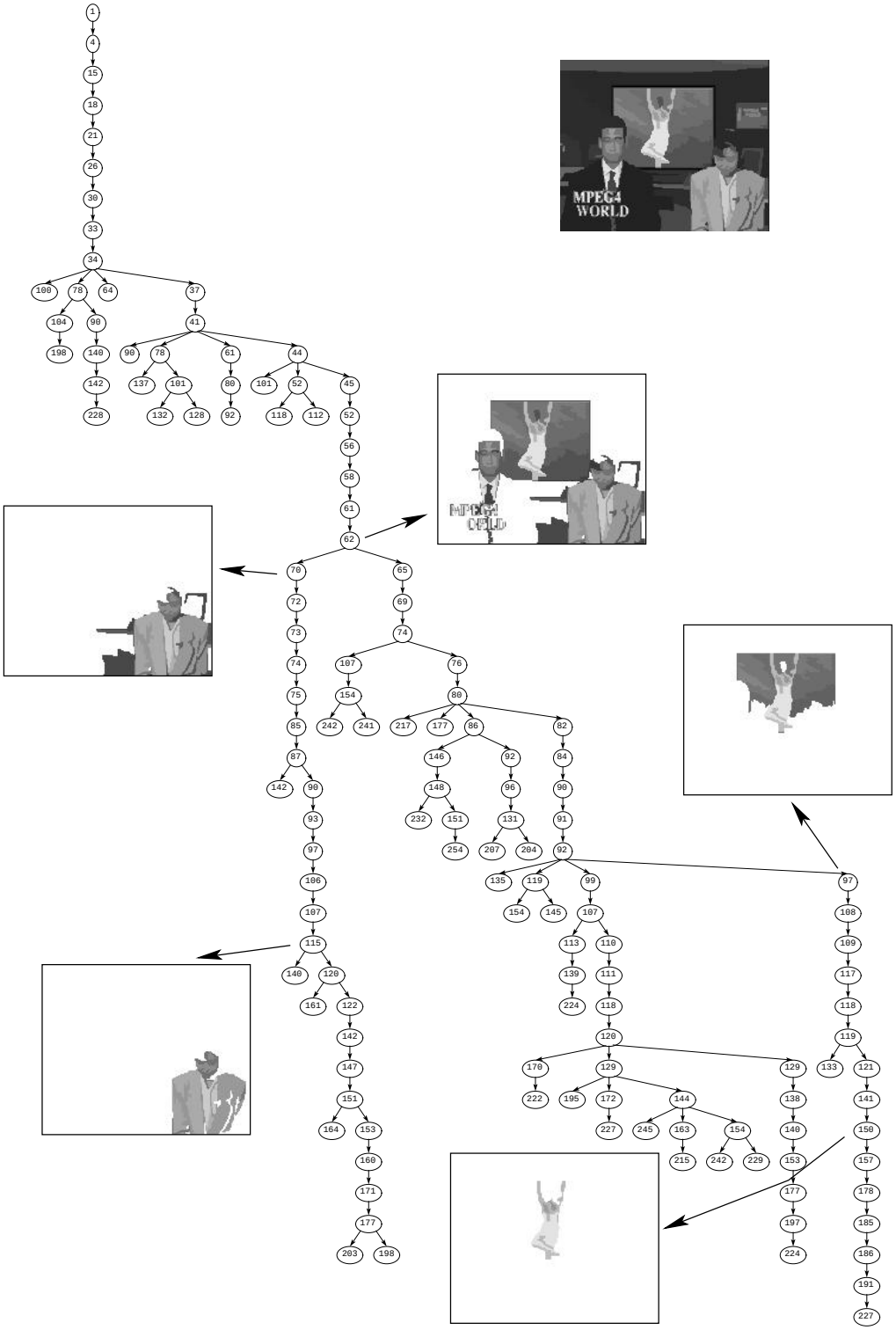


Figure 3.3: Max-Tree example. The original image is shown on top. At each node of the Max-Tree, its associated gray-level is indicated.

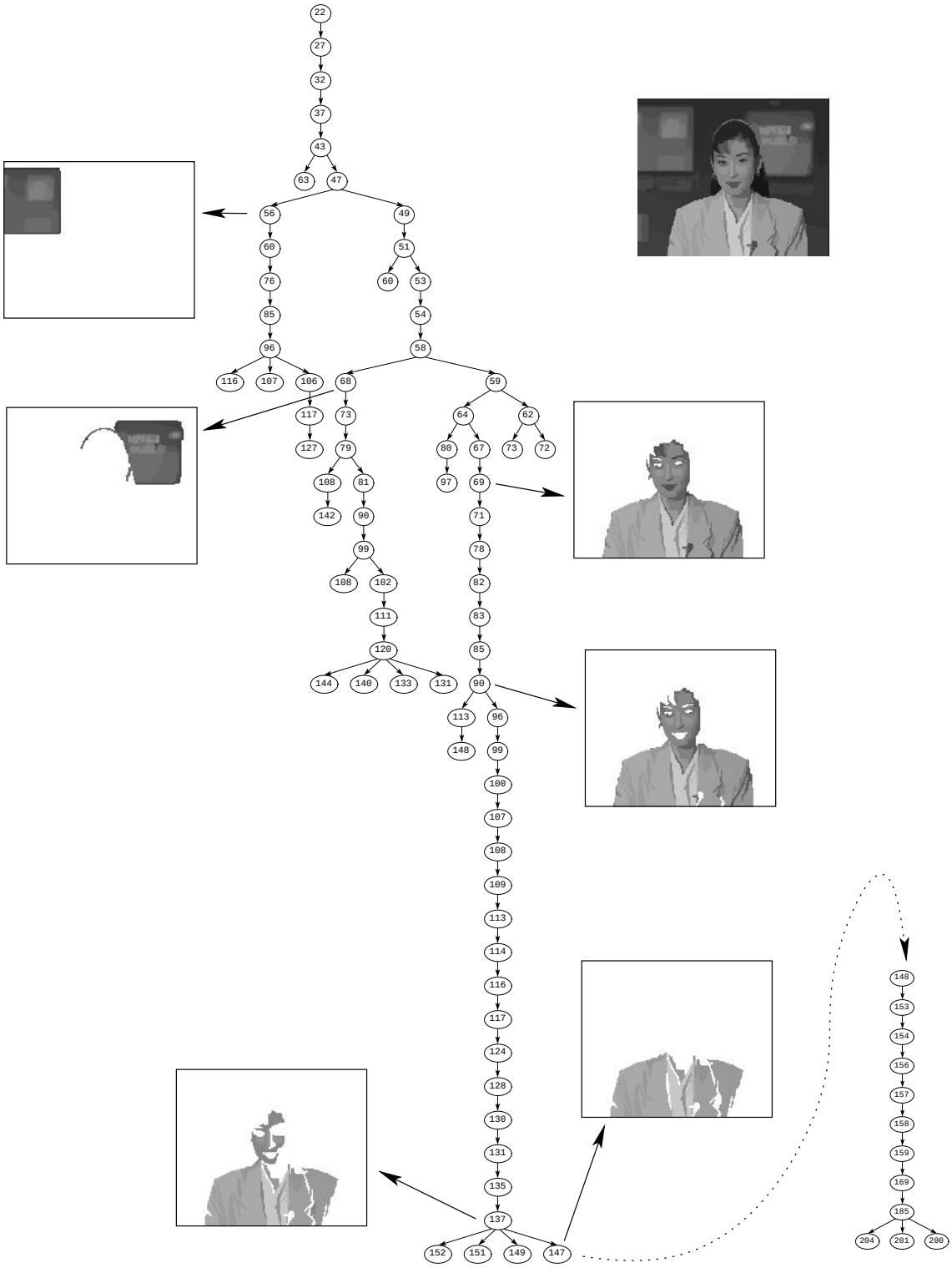


Figure 3.4: Max-Tree example. The original image is shown on top. At each node of the Max-Tree, its associated gray-level is indicated.

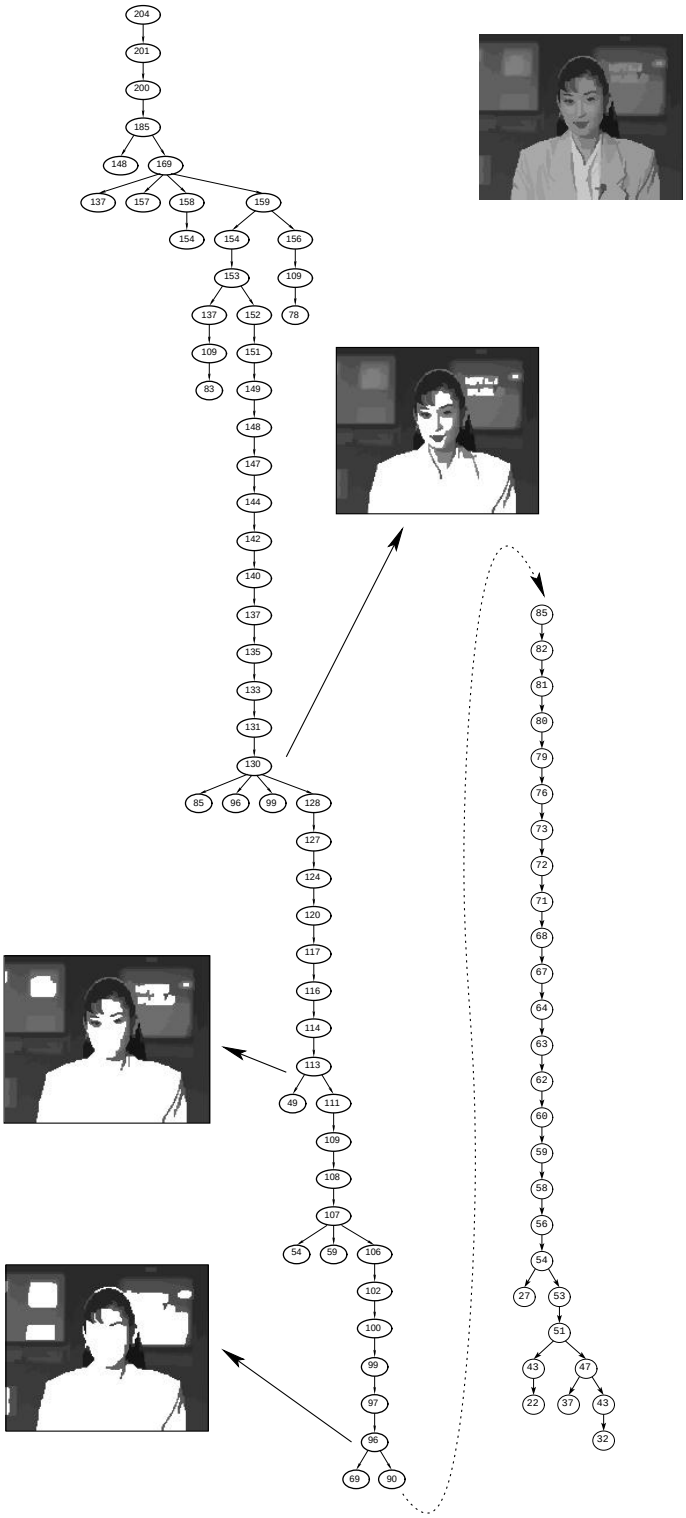


Figure 3.5: Example of Min-Tree representation. The original image is shown on top. At each node of the Min-Tree its associated gray-level is indicated.

3.2 Min-Tree

By duality, the Min-Tree is defined. Whereas the Max-Tree is constructed using the upper level sets, the Min-Tree is a structured representation of the lower level sets. As for the Max-Tree, in the Min-Tree each connected component of $X^h(f)$, $LowCC_k^h$, is associated to a node of the tree. The parent relationship is established based on the inclusion relationship of the connected components of $X^h(f)$ and $X^{h-1}(f)$. The Min-Tree is a structured representation of the image oriented towards the minima of the image. The leaves of the tree represent the regional minima of the image, and the root node corresponds to the greatest gray-level value in the image. In Fig. 3.5 an example of Min-Tree is shown.

The Min-Tree can also be constructed on $-f$ using the Max-Tree construction algorithm. In the image processing framework, $-f$ is usually computed as $NG - f$, where NG represents the highest possible gray-level value of an image (usually $NG = 255$).

3.3 Discussion

The Max-Tree and Min-Tree structures are scale-space representations. Connected components of small size appear near the leaf nodes, whereas large size connected components appear near the root node. Moreover, there is a close relationship of the Max-Tree and Min-Tree with the Area Tree and Inclusion Tree reviewed in Sec. 2.3.4 and Sec. 2.3.5, respectively.

First, notice that the Max-Tree can also be constructed by applying the algorithm described in 2.3.4 using the area opening γ as sieve. In fact, let us denote with γ_s the area opening of size s [94]. When γ_s is applied on the image, the regional maxima of size s are removed from the image. The area opening of size s applied over an image f ensures that in the resulting image $g = \gamma_s(f)$ all of its connected components of the upper level set have a size greater (or equal) than s [94]. The Area Tree created using γ_s as sieve represents the way the connected components are removed from the image as scale s increases. Small connected components appear near the leaves of the tree since they are the first to be removed, whereas large connected components appear near the root node. The way the area opening γ_s acts on the image is represented in fact by the Max-Tree (we assume that its associated empty nodes have been removed, see example in Sec. 3.1).

The reader may have noticed, by comparing Fig. 3.4 and Fig. 3.5 that the structures of a Max-Tree and of its dual, the Min-Tree, are quite different. In [43] (see also Sec. 2.3.5) this issue is studied and a new representation is developed, the Inclusion Tree, in which the information associated to the Max-Tree and Min-Tree has been merged into one tree. Moreover, the Inclusion Tree representation is self-dual. That is, f and $-f$ lead to the same tree representation.

Pruning algorithms applied to a Max-Tree result in removal of bright components of the

image, while dark components are removed if the pruning is applied to a Min-Tree. In other words, each tree is oriented towards the removal of different types of components of the image. The resulting operators are called *anti-extensive* and *extensive*, respectively. Pruning operators on these trees are not self-dual. Self-duality may be an important issue depending on the application, since it ensures that bright and dark components are processed in a symmetrical way. In Chap. 4 we will see that the Binary Partition Tree may be constructed in a self-dual manner, and therefore the resulting pruning strategies result in self-dual operators. As a matter of fact, the Inclusion Tree reviewed in Sec. 2.3.5 leads to self-dual operators.

3.4 Efficient implementation

3.4.1 Algorithm

The efficient implementation of the construction of the tree is based on a recursive flooding algorithm. The flooding algorithm begins at the root node of the tree (that is, the lowest gray value of the image f) and constructs in a recursive way each of the branches of the tree.

Our algorithm is based on using hierarchical first-in first-out (FIFO) queues. First-in first-out queues have been extensively used in order to implement operators such as the watershed or reconstruction with a marker [95, 94]. We need $NG + 1$ first-in first-out queues, where NG is the highest possible gray-level value of an image f (usually $NG = 255$). Each of the individual queues is associated to a particular gray-level value h. These queues are used to define the scanning and processing order of the pixels composing the image. The processing is done in such a way that each pixel of the image needs to be introduced only once in one of the first-in first-out queues. The set of queues are used in a hierarchical way. Pixels associated to the queue with highest gray value are processed first.

In order to create the Max-Tree, the following three queue functions are necessary

- **fifo-add(h,p)**: add the pixel p (of gray-level h) in the queue associated to gray-level h.
- **fifo-first(h)**: extract (and remove) the first available pixel of queue of associated to gray-level h.
- **fifo-empty(h)**: returns "true" if queue associated to gray value h is empty. Returns "false" otherwise.

We will also make use of the following notations: **number-nodes(h)** defines the number of nodes that have been found so far at gray-level h. The values of number-nodes are initialized to zero at the beginning of the tree construction. As the tree structure is created, this variable is updated as new nodes are created at gray-level h. The variable **node-at-level(h)** is used to know if, for the branch that is being analyzed, a non-empty node is located at gray-level h. Its values are initialized to "false".

Let ORI denote the original image and $ORI(p)$ the gray-level value of pixel p. The matrix $STATUS$ stores the information of the pixels; $STATUS(p)$ gives us the state of a particular pixel p: "not-analyzed" if the pixel p hasn't been visited by the algorithm, "in-the-queue" if the pixels p has been introduced in the queue, or $STATUS(p) > 0$ indicating that pixel p belongs to the node $UpCC_{ORI(p)}^{STATUS(p)}$. All components of the $STATUS$ matrix are initialized to "not-analyzed".

The flooding procedure is described precisely in Fig. 3.6. The pseudocode can be divided into two stages: the first one (lines 02–14) actually performs the propagation of the pixel through its associated flat zone, whereas the second step (lines 15–26) defines the parent/child relationships. The recursive call is located in line 13. Note that line 05 in Fig. 3.6 defines the neighborhood relationship between two pixels, and therefore, defines the connectivity of the connected components of each node $UpCC_h^k$. Supported connectivities are based on 4 and 8 neighbor relationship.

The flooding algorithm is rather complex, and therefore let us see how the Max-Tree is created for the image in Fig. 3.2. For that purpose, lets denote with $\text{flood}_N(h)$ that the algorithm is in the N'th level of recursivity. The tree begins to be created by first locating in the image a pixel with the lowest gray-level value $h = h_{min}$, then inserting such pixel in the queue at level h_{min} and calling $\text{flood}_0(h_{min})$. Any pixel of the flat zone A may be therefore be inserted in the queue at level $h = 0$. Then, $\text{flood}_0(0)$ begins the flooding through the flat zone A (lines 05–14 in Fig. 3.6). Assume that during the flooding a pixel of flat zone C is found. This pixel is inserted in the queue at level 2 (line 08) and then a recursive call, $\text{flood}_1(2)$, is performed (line 13). The flooding through region C is then performed. Note that at is point we still haven't found any pixel of flat zone B, D or F (which is the parent of node associated to flat zone C, see Fig. 3.2). During the flooding of region C, some pixels of region B or D are found. These pixels are inserted in the queue at level $h = 1$ and node-at-level(1) is set to "true", indicating that there is a node at level $h = 1$ in the branch of the tree that contains region C (lines 08–10). Note that during the analysis of region C no recursivity call is done (that is, the current node being analyzed has no descendants). When the flooding of region C is finished (that is, the FIFO queue associated to gray value $h = 2$ is empty, line 02), the code passes to search for (the gray value of) the parent of region C (lines 15–17). Since node-at-level(1) is "true", the loop at line 16 ends with $m = 1$. Line 21 establishes the parent relationship between the two nodes, that is, the node $UpCC_1^1$ (associated at the moment to some pixels of flat zone B and D) is the parent of node $UpCC_2^1$ (associated to flat zone C). The function returns $m = 1$ (that is, the gray value associated to the parent that has been found), and this value is assigned to m at line 13 of the previous call, $\text{flood}_0(0)$, in which region A is analyzed. The code then checks (line 14) if the parent that that has been found at the previous recursive call (in this case $UpCC_1^1$) is the same than the region that is being flooded at that moment (in this case $UpCC_0^1$). For that purpose, only the gray-level associated to the nodes has to be checked. In our case, $m = 1$ and $h = 0$, and therefore

the loop condition (line 14) is not hold. A recursive call is then performed (line 13) with $m = 1$, $\text{flood}_1(1)$. The propagation of flat zones B and D is then performed (remember that previously, during flooding of region C, some pixels of B and D where introduced in the queue). During such propagation, some pixels of A may be introduced in the queue at level $h = 0$. Note that some pixels of E should also be found. As before, a new recursive call is then performed to define the region of support of E. During this propagation, some pixels of F may be introduced in the queue at level $h = 1$ and as before, a new parent relationship is defined (lines 15–23) indicating that the node $UpCC_1^1$ is parent of node $UpCC_2^2$ (which is associated to region E). The function $\text{flood}_2(2)$ then returns $m = 1$, and the code returns to the point of the previous recursive call, $\text{flood}_1(1)$. The condition at line 14 is hold, and then the propagation of flat zones B, D and F is continued until the queue at level $h = 1$ is empty (line 02). Note that once the queue is empty, the support of node $UpCC_1^1$, which is made of flat zones B, D and F, has been defined. Then the algorithm defines the parent for node $UpCC_1^1$, which is node $UpCC_0^1$, and returns a value of $m = 1$ to $\text{flood}_1(0)$. The condition at line 14 is hold, and the propagation of flat zone A is continued. During this flooding, the flat zone G should be found and defined with a recursive call. Finally, when $\text{flood}_0(0)$ ends with an empty queue (line 02), the tree has been completely defined. The condition at line 18 is defines node $UpCC_0^1$ as the root node (line 23).

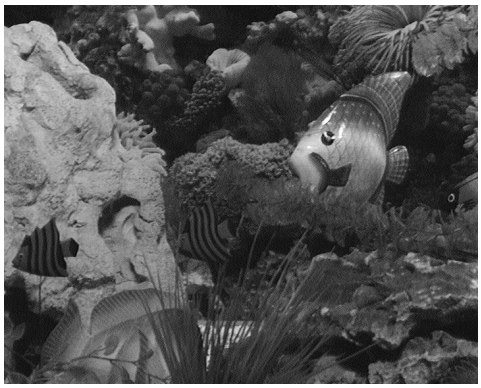
3.4.2 Tree complexity

Several tests have been performed in order to estimate the complexity of the tree. Since the tree is oriented to represent gray-level images, its associated *height* (see Sec. 2.2.1) is at most NG. Thus, the number of nodes of the tree representation created from an image may be used as a measure of the tree complexity. Fig. 3.7 shows the two images that have been taken to perform our tests. In order to assess the number of nodes for different image sizes, we have taken squared subimages of different size by placing the upper left corner of these subimages at all possible positions given by a fixed squared grid. The points of the grid are spaced 10 pixels. Fig. 3.8 shows the plot of number of nodes of the tree representation as a function of the image width (or height) in pixels against of the subimages. As expected, the number of nodes of the resulting tree depends on the image content. For instance, for an image width of 250 pixels, the number of nodes in the tree structure ranges from 1.45×10^4 to 1.7×10^4 and from 1.9×10^4 to 2.3×10^4 for the plot shown respectively on top and bottom of Fig. 3.8. It can be seen that for larger sized images (550 pixels) the number of nodes of the tree increases to values which are greater than 10^5. As can be seen, the number of nodes of the Max-Tree representation may be obtained.

```
01  flood(h)
02      while not fifo-empty(h)
03          p ←fifo-first(h)
04          STATUS(p) ← number-nodes(h) + 1
05          for every q ∈ N_E(p)
06              if STATUS(q) = "not-analysed"
07                  m ← ORI(q)
08                  fifo-add(m, q)
09                  STATUS(q) ← "in-the-queue"
10                  node-at-level(m)← "true"
11                  if (m > h)
12                      repeat
13                          m ← flood(m)
14                      until m = h
15      m ← h − 1
16      while m ≥ 0 and node-at-level(m) = false
17          m ← m − 1
18      if m ≥ 0
19          i ← number-nodes(h) + 1
20          j ← number-nodes(m) + 1
21          UpCC_m^j is parent of UpCC_h^i (UpCC_m^j → UpCC_h^i)
22      else
23          UpCC_h^{i=1} has no father (UpCC_h^{i=1} is root node)
24      node-at-level(h)← "false"
25      number-nodes(h)← number-nodes(h) + 1
26      return m
```

Figure 3.6: Flooding procedure for the creation of the Max-Tree.

Size 352×288



Size 720×576

Figure 3.7: Images used to test the tree construction algorithm of Fig. 3.6.

3.4.3 Performance

Number of comparisons

The performance of the proposed algorithm (see Fig. 3.6) is rather high, since each pixel needs to be introduced only once in the hierarchical queue. Thus, each pixel is processed (lines 05–14 of algorithm code) only once. Each time a pixel is extracted from the queue it is compared with its neighbors. When 4-connectivity is used the algorithm performs 4 comparisons for each pixel, and when 8-connectivity is used a total of 8 comparisons are performed. This is true even for the pixels located at the frontier of the image, since before entering the construction algorithm a border is set around the image avoiding thus the condition statements that would be needed in order to know if we are accessing a pixel outside the image. Therefore, for an image of size $N \times M$, the total number of comparisons that are performed is $N \times M \times 4$ or $N \times M \times 8$ depending on if 4 or 8 connectivity is used, respectively. Note that the number of comparisons is constant for a specific image size and does not depend on the image content. This does not mean, however, that the CPU time needed to construct the tree does not depend on the content on the image. Note that as new connected components are located in the image, new nodes have to be dynamically created in the memory heap.

CPU time

On of the most usual figures used to test the performance of an algorithm is the CPU time, that is, the number of seconds the algorithms spends to perform a specific task. In our case, we have tested our algorithm using a Pentium II processor running Linux with a clock speed of 400MHz and 256Mbytes of RAM.

Fig. 3.9 shows the time needed to construct the tree for the subimages of Fig. 3.7. Similarly

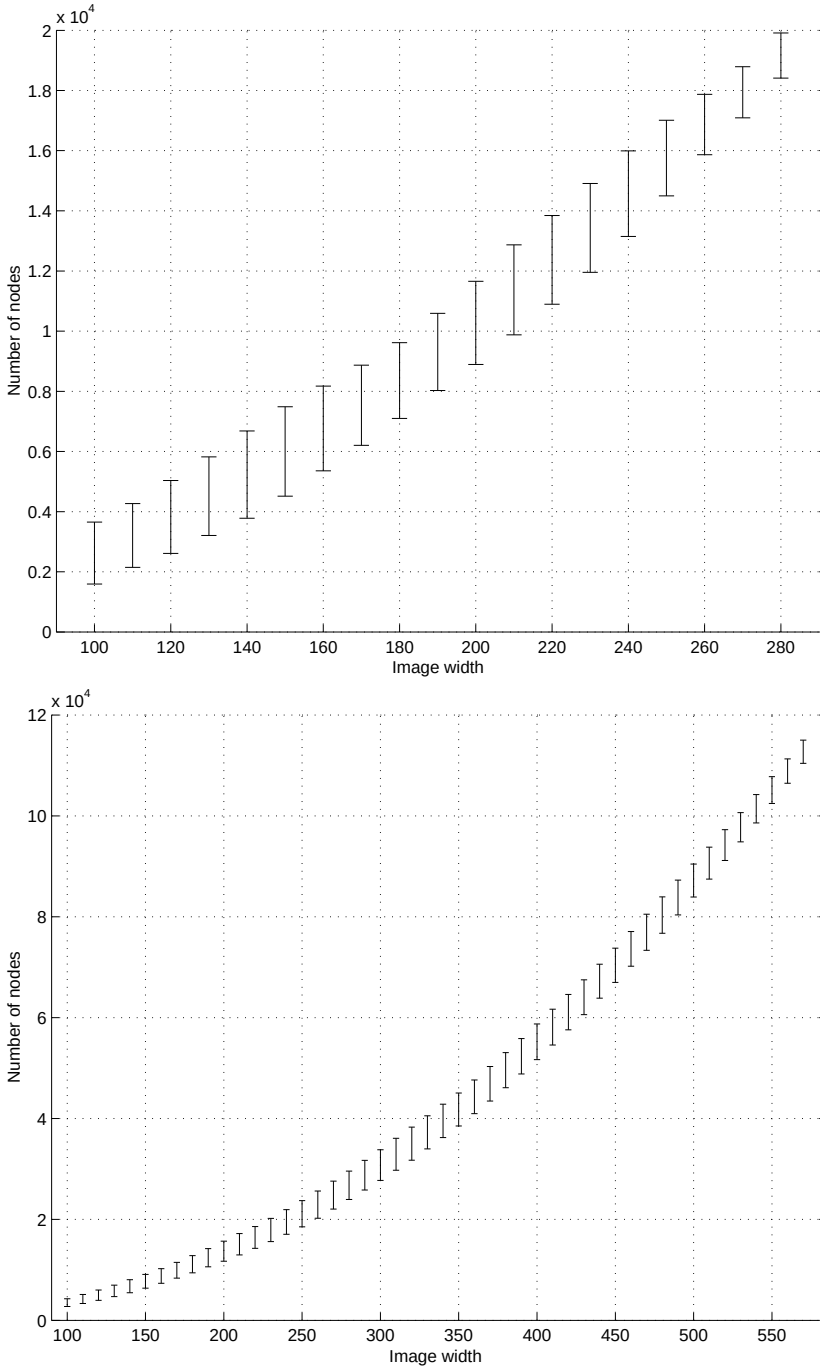


Figure 3.8: Number of nodes for several squared subimages. The maximum and minimum obtained value is indicated. Top: Original image shown on the left of Fig. 3.7, Bottom: Original image on right of Fig. 3.7.

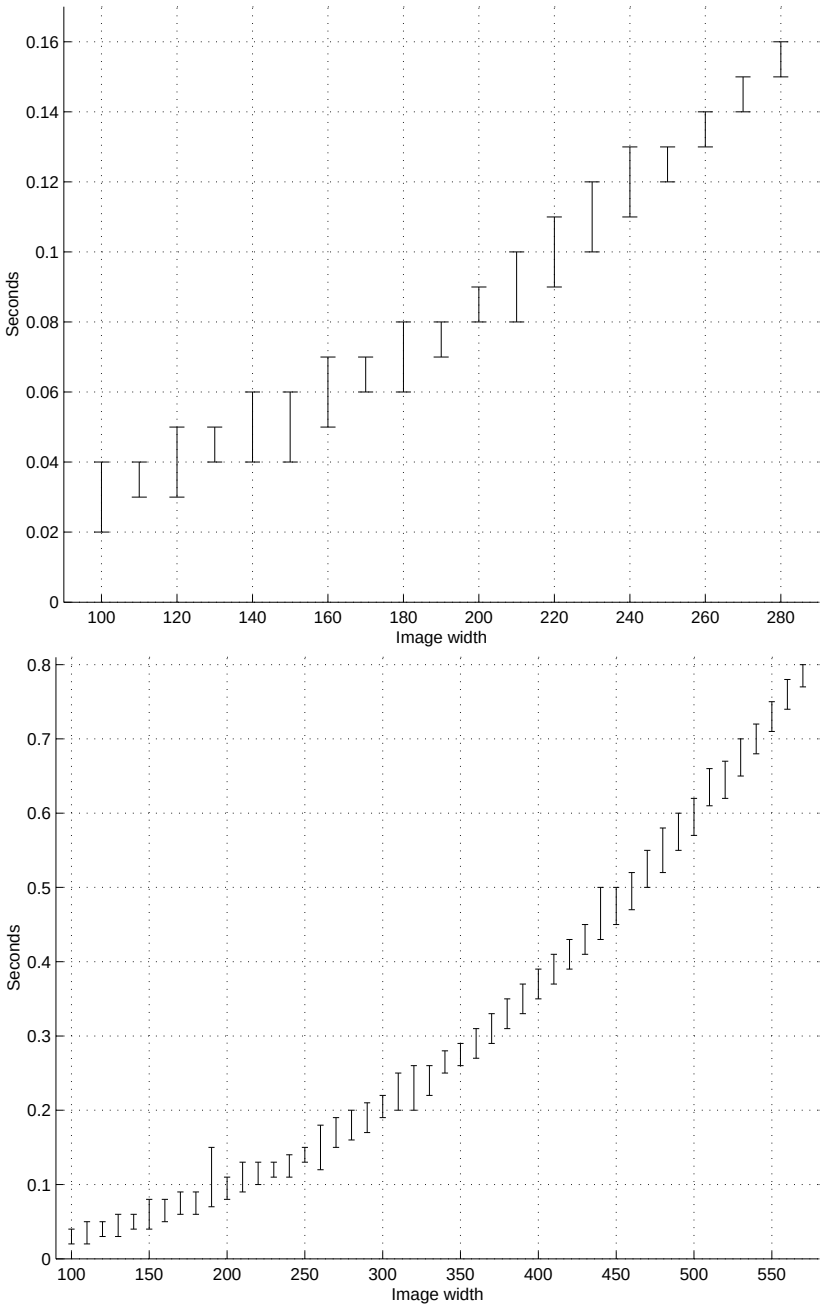


Figure 3.9: Time spent for the tree construction for several squared subimages. The maximum and minimum measured value is indicated. Top: Plot for the image is shown on the left of Fig. 3.7, Bottom: Plot for the image on right of Fig. 3.7. Tests performed on a Pentium II 400Mhz based sytem.

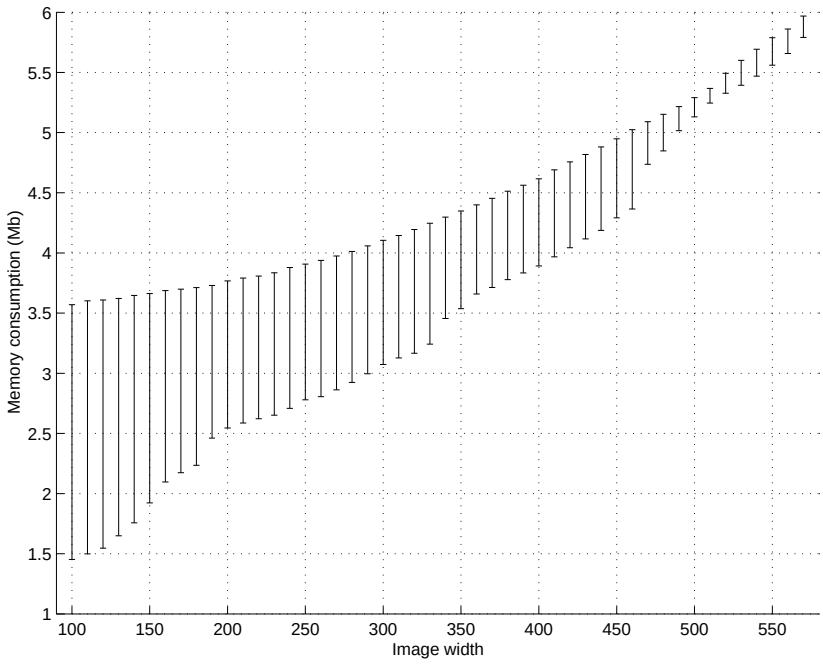


Figure 3.10: Memory consumption, in Megabytes (Mb), for the tree construction algorithm using as input several squared subimages of the image shown on the left of Fig. 3.7.

to Sec. 3.4.2, for a fixed image size the time needed to construct the tree depends on its contents. The results show that the algorithm is very efficient for mid size images: less than 0.8 seconds for images of size lower than 500×500 is enough for enabling the use of this representation for a wide range of applications.

Memory consumption

An important figure used to test the algorithm performance is the memory consumption, which is plotted in Fig. 3.10. As for the CPU time, memory consumption depends on the contents of the image. As can be seen, memory consumption is kept to rather low levels. Moreover, it is worth to mention that the actual implementation of the tree structure construction has not been designed to be efficient in memory. For instance, in our node structure definition we have a member that points to the parent and a member that points to the children nodes. This is a redundant representation since it is enough to keep only the parent relationship. Thus, the plot shown in Fig. 3.10 should be considered as a rough upper bound of the memory usage.