

Efficient Algorithms to Implement the Confinement Tree

Julian Mattes^{1,2} and Jacques Demongeot¹

¹ TIMC-IMAG, Faculty of Medicine
38700 La Tronche, France

Julian.Mattes@imag.fr, Jacques.Demongeot@imag.fr

² iBioS, DKFZ Heidelberg
69120 Heidelberg, Germany
J.Mattes@dkfz.de

Abstract. The aim of this paper is to present a new algorithm to calculate the confinement tree of an image – also known as component tree or dendrone – for which we can prove that its worst-case complexity is $O(n \log n)$ where n is the number of pixels. More precisely, in a first part, we present an algorithm which separates the different kinds of operations – which we call scanning, fusion, propagation, and attribute operations – such that we can separately apply complexity analysis on them and such that all operations except propagation stay in $O(n)$. The implementation of the propagation operations is presented in a second part, first in $O(n_n^2)$, where n_n is the number of nodes in the tree ($n_n \leq n$). This is sufficient if the number of pixels is much larger than the number of nodes ($n_n \ll n$). Else, we show how to obtain $O(n_n \log n_n)$ complexity for propagation. We construct two example images to investigate the behavior of two known algorithms for which we can show worst-case complexity of $O(n^2 \log n)$ and $O(n^2)$, respectively, and we compare it to our algorithm. Finally, a practical evaluation will be opposed to the theoretical results. Several variations of the implementation will show which operations are time consuming in practice.

1 Introduction

In the last years, the confinement tree proved to be a useful tool in image processing. This summary of an image in which each node corresponds to an image region (called a *confiner*) has been (re)discovered by several authors according to various applications in image processing, but its origins lie in statistics [14,6]. Applications have been image filtering, segmentation [5,4,3,7], matching [10,9,12,11], and classification [10], object detection [5] and recognition [10,12]. Structures similar to the confinement tree are investigated in [13]. Note that especially when we are concerned with 3D imaging the time to calculate the tree (or the trees for each of the 2D slices forming the 3D image) becomes prohibitive. For 3D reconstruction in X-ray audio-graph imaging we have to align (match) up to 1000 2D image slices and to calculate for each one, as well as for

the whole 3D image, the confinement tree for image matching [11]. Fast execution time is also required for segmentation of cell regions in high-throughput scanning [15].

In this paper we want to present and investigate the difficulties that appear when looking for an efficient implementation of the confinement tree and to propose possible solutions. There is a need for such investigations because even if for some applications very fast execution times are required, in a first approach, people focus more on the ability of the algorithm to solve the studied problem than on its optimization. For instance, Guillataud [4] proposed an implementation which determines the confiners successively and independently at each level and [5,3] made only suggestions for an algorithm which “may be in almost-linear time” [3] based on immersion simulation. Jones [7,8] made a first attempt to consider also the optimization problem. He obtained in general much faster execution times but for certain images (as shown on theoretical examples in section 3) we can considerably improve the fastness. In worst-case we can improve the complexity by passing from $O(n^2 \log n)$ to $O(n \log n)$. Salembier et al. [13] presented already before an efficient algorithm to calculate the max-tree which is a tree related to the confinement tree (see section 3). We apply similar basic techniques as Jones in [8] but our global strategy is different. In addition we will give more details (sections 3.3 and 3.2) and provide a complexity analysis in section 4. Our experimental evaluation in section 5 will show where to achieve more efficiency in practice. Before presenting our algorithm in section 3 we give first definitions and notations in section 2.

2 Definitions and Notations

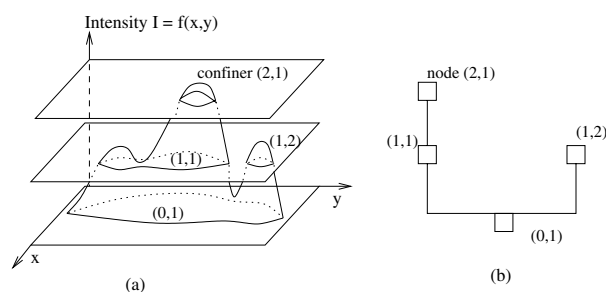


Fig. 1. Illustration after [4] for (a) the definition of the confiners and (b) the confinement tree

Given an intensity function $f : \mathbb{R}^d \rightarrow \mathbb{R}_{\geq 0}$, $d = 2, 3$, the *confiners* are defined as the maximal connected subsets (i.e., *components*) \mathcal{C}_l of the level sets $\mathcal{L}_l = \{x \in \mathbb{R}^d | f(x) \geq l\}$, $l \in \mathbb{R}_{\geq 0}$ (see Figure 1(a)). Considering them taken on several levels l_k , $k = 1, \dots, r$ (r from resolution) including the 0 level,

they define obviously a tree (by “set inclusion”; see Figure 1(b)) called *confinement tree*. In practice, the levels are those of the image’s grey intensity function and we take all available grey levels into account. If we do not represent identical confiners in the tree but only the highest among them (see Figure 2) we call this the *unique representation*. We choose the discrete $d4$ – distance for defining the connectivity between pixels (see [7] for illustrations).

We found these components (confiners) and tree first in the classification domain in a paper by Wishart [14] and further investigations by Hartigan [6], where the confiners are called *high density clusters*. In image processing these sets and this structure appear first in [5]. See [14,5,10,11] for the invariance properties of the confinement tree. Another kind of components, induced in a natural way by the grey level function, are the connected components of the sets $\{x \in \mathbb{R}^d | f(x) = l\}$ [13]. They are called *flat regions*. We will use throughout this paper the notations n for the number of pixels in the image, n_n for that of nodes in the tree, and r for the number of grey levels. We assume $r \leq n$.

3 Implementation

When designing an efficient algorithm implementing the confinement tree, there arise some difficulties from the necessity to scan at each grey level only pixels bringing a new knowledge about the tree and to propagate information about connectivity of regions in the image. Hereby, we will distinguish in the following *attribute operations*, applied to update the tree attributes, and *connectivity operations* which are necessary to find the connection from a pixel to a confiner and to establish the father-son relationship in the tree. They include *scanning*, *fusion*, and *propagation operations*, the latter are needed to propagate neighborhood information through the already updated part of the tree.

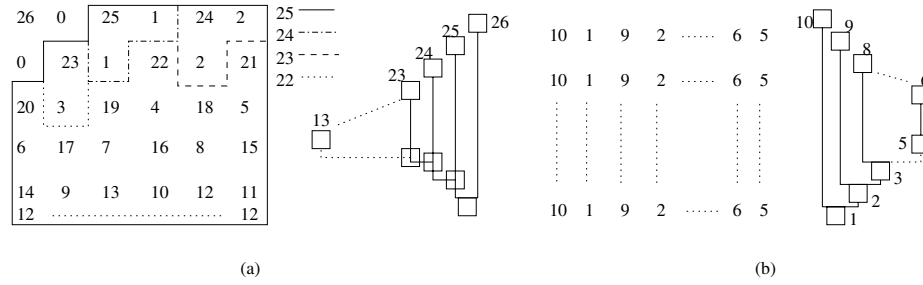


Fig. 2. Example images: (a) for worst case analysis for the algorithms of Jones [8] and Salembier et al. [13]; (b) for comparison with our algorithm. On the corresponding confinement trees we have chosen unique representation of identical confiners. We assume 4-connectivity

Algorithms to calculate confiners for a given grey level in linear time with respect to n are well known [1]. However, when we look for an algorithm which calculates the confiners for *all* grey levels but stays within $O(n)$ attribute operations and $O(n \log n)$ connectivity operations there arise some complications making it worth to be detailed.

A first attempt at designing an efficient algorithm tackling this problem was made by Jones [7,8]. His algorithm calculates first all local maxima of the image's grey level function (i.e., the leaves of the tree). After it detects, starting from each local maxima in sequential order, the confiners at the different grey levels on the path (called *branch* [8]) between the leaf and the root which are not already updated. For each leaf m the k_m pixels of these confiners are updated using a priority data queue in which they are ordered according to their grey value. In Figure 2(a) this will be done, for a given leaf m , in $O(k_m \log k_m)$ time. The pixels associated to the leaf at level 25 are delimited by the continuous 25-line. Pixels in not updated confiners on the branch between leaf 24 and the root are confined by the 24-line and the longer part of the 25-line, etc. Therefore, if – as in Figure 2 – many isolated local maxima with high grey level appear in the image the algorithm examines multiple times large parts of it, even if n_n is small as depicted in Figure 2(b) (see also section 5). The worst case behavior (Figure 2(a)) is $O(n^2 \log n)$. If we try to overcome the problem by stopping this detection process for a given leaf as soon as the grey levels increase we have to pay attention to the fact that in this case two higher branches (as 23 and 24 in 2(a)) can be separated by lower ones (as 19 and 22). We would need procedures as we will propose in steps (B2 l) and (B3 l).

Another approach for calculating the confinement tree would be to add attribute operations to the max-tree creation procedure and then to derive the confinement tree from the max-tree. Salembier et al. [13] presented an elegant algorithm to calculate the max-tree and achieved very fast results in practice. Applied to the image of Figure 2(a) however, their algorithm runs in $O(n^2)$ time. As their paper did not address principally to the optimization problem, a detailed evaluation is not given.

In the following, we describe first globally the algorithm, after we give detailed information for procedures implementing it, as presented in the Figures 4, 5, 6.

3.1 Global Strategy

The discrete connectivity model we have chosen is that of 4-connectivity (for illustration see [7]). The pseudo-code used below is based on ANSI-C.

Data Structure of the Result to be Determined Our algorithm aims to calculate the components of the following data structure: for each confiner (node) we keep in a structure (**struct**) (1) its surface, (2) its mass, (3) the X and the Y coordinates of its gravity center, (4) the eastmost among its northmost pixels (to be able to recalculate easily its contour), and (5) a number for identifying the node of its father in the array **Tree** defined next. The tree is implemented

then as a 2 dimensional (dynamic) array **Tree**. It associates to a given level l (in the tree) and to the given number i of a node at this level the structure of the corresponding confiner (node) C_{il} . C_{il} is also at level l in the image F (in the procedures we will use the notation L for the level). In an additional array **NodeNumb** we keep the number of nodes at the different levels. In the description of our implementation we will first represent identical confiners (see Figure 3(c), confiner 4 at the levels 10, 11, 12) at each new level again. After, we will show what is to change when representing only the confiner at highest level among identical confiners (“unique representation”). This is necessary for worst case analysis.

Principal Steps of the Algorithm The algorithm proceeds in two steps: first (A) the pixels are sorted in decreasing order according to their grey level. The order at the same level is arbitrary. This is done by a standard sort algorithm called “counting sort” [2] in $O(n+r) = O(n)$ (we assume $r \leq n$) time. In the second step (B) we calculate the confiners for all grey levels l (step B l) in decreasing order, beginning with the highest grey level, and we update the tree structure up to the given level. Step (B l) consists of four sub-steps (B1 l), ..., (B4 l). Step (B1 l) (Figures 3(a), 4) implements the *scanning* operations and determines the flat regions at level l . For each pixel the number of the corresponding flat region is provisionally kept in memory (in **Pix2cf**, see below). Step (B1 l) determines also which already updated confiners (obtained using **Pix2cf**) – and even their ancestors at level $l+1$ using the data structure maintained in step (B3 m), $m > l$ – are adjacent to such a region. This information is kept in memory (in **Fusion**, see below). We deduce from it in step (B2 l) (Figures 3(b), 5) which confiners at higher levels and which flat regions at level l are fusing for obtaining the new confiners at level l (stored then in **Pix2cf**). Step (B2 l) implements the *fusion* operations.

That we have to know (in constant time if possible) for a given confiner its ancestor at level $l+1$ (Figure 3(a),(c)), in order to fuse the confiners and flat regions, is one of the major difficulties (with respect to worst-case behavior) of the confinement tree construction. To do this, in step (B3 l), detailed in section 3.3, we define and maintain (by the *propagation* operations) a data structure associated to the tree. It is called **Node2ances** and replaced in the optimized versions of (B3 l) (cf. section 3.3) by **Leaves2ances** or by **SubtreeL2ances**. Finally we pass in step (B4 l) once more through all pixels at level l to update (by the *attribute* operations) the attribute values of the now identified confiners (Figure 6).

Step (B3 l) is crucial for worst-case analysis. In section 3.3 we present an algorithm (Figure 7) to execute step (B3 l) in $T_3(l)$ time such that $\sum_l T_3(l) = O(n_n \log n_n) = O(n \log n)$ time, assuming unique representation of identical confiners. Steps (Bi l), $i = 1, 2, 4$, are executed in $\sum_l T_i(l) = O(n)$ time by the procedures presented in Figures 4, 5 and 6. However, as we have in imaging practice often $n_n \ll n$, step (B3 l) will not in general be the most consuming one (cf. section 5).

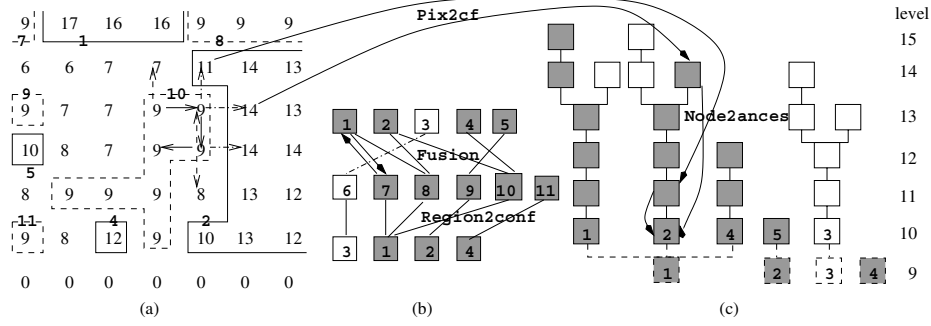


Fig. 3. (a) Part of an image to illustrate the detection of flat regions and confiners for a given grey level: dashed lines delimit flat regions at the current level 9 and continuous lines delimit confiners at level 10; the numbers on the lines are that of the corresponding confiners or regions respectively; different arrow types correspond to different cases in Figure 4 (see section 3.2). (b) Boxes contain confiner numbers at level 10 (upper row) and 9 (lower row) and flat region numbers (middle row). Broken lines and (as in (c)) white boxes correspond to not shown parts of the image. Arrows illustrate (as in (c)) the association made by the stated data structures; we omit in **Region2conf** implicitly given associations between the confiners at levels 9 and 10. (c) Confinement tree updated up to level 9; broken lines illustrate information added in step (B 9)

Four Principal Data Structures The sub-steps (B1 l), ..., (B4 l) are related by four data structures essential for our algorithm. There are (cf. Figure 3) **Pix2cf**, **Fusion**, **Region2conf**, and, in a first approach, **Node2ances** which will be replaced by **Leaves2ances** or by **SubtreeL2ances** in section 3.3 when showing how to pass from $O(n^2)$ complexity to $O(n \log n)$ for (B3 l). **Fusion** is a dynamic array of stacks (see below); the other structures are dynamic arrays of integer values. Let us detail now, how this structures are defined and cooperate during the sub-steps (B1 l), ..., (B4 l).

To each flat region at level l , determined in step (B1 l), we allocate a number $R_{i'l}$ which we store provisionally in each of its pixels (in an array **Pix2cf**). Equally, we store for each (already discovered) confiner $C_{j(l+1)}$ at level $l+1$ adjacent to such a region this number $R_{i'l}$ in a stack **Fusion**($C_{j(l+1)}$) and vice versa $C_{j(l+1)}$ in a stack **Fusion**($R_{i'l}$). See in step (B3 l) and in Figure 3(a),(c) how we find the number of $C_{j(l+1)}$. The stacks in **Fusion** permit (as illustrated in Figure 3(b)) to connect in step (B2 l) all flat regions $R_{i'l}$ and all confiners $C_{j(l+1)}$ at the previous level which belong now to the new confiner C_{il} at the current level l . Therefore, these connections enable us also to establish the father-son relationships between the confiners (nodes) at two successive levels. **Fusion** is implemented as an array containing after step (B1 l) the stacks **Fusion**($C_{j(l+1)}$) for all confiners $C_{j(l+1)}$ at level $l+1$ and the stacks **Fusion**($R_{i'l}$) for all flat regions $R_{i'l}$ at level l . Using **Fusion** we merge in step (B2 l) all connected

regions in order to obtain the confiners at level l . Here, we associate in the array **Region2conf** (see Figure 3(b)) each number identifying one of the confiners $C_{j(l+1)}$ at level $l+1$ and each flat region number $R_{i'l}$ with the number of the new confiner C_{il} containing the given confiner or region. Using **Region2conf**, we replace in the array **Pix2cf** the flat region number $R_{i'l}$ by that of the confiner C_{il} enclosing it. This is done by the procedure in Figure 5.

Step (B3 l) operates on the tree only. We update an array **Node2ances** (in a first approach: more sophisticated approaches to implement step (B3 l) are proposed in section 3.3; they do not require changes in the other parts of the algorithm). **Node2ances** maps (Figure 3(c)), before step (B3 l), each already determined node to its ancestor at level $l+1$ and to its ancestor at level l , after (B3 l). This will be necessary for the steps (B1 $l-1$) and (B2 $l-1$): we stored (in the array **Pix2cf**) in the steps (B2 m), $m \geq l$, for each pixel at level m the number of the node/confiner at level m containing it but we will need its ancestor node at level l . In step (B4 l), finally, we examine once more all pixels at level l . We add to each new node first the attribute information coming from its sons in the tree by the procedure **addNode(node, prevNode)** and after the attribute information coming from the pixels by **addPixel(node, pixel)**. The corresponding procedure is given in Figure 6.

3.2 Our Algorithm in Detail

Let us give now some more detailed information for the procedures presented in the Figures 4, 5, 6 and show some techniques to speed up the algorithm.

```

1. init(Fusion); /* initial. Fusion as array of empty stacks */
2. RegNumber = NodeNumb[L+1];
3. for each regElement at level L do
4.   if (Pix2cf[regElement] == 0) /* new flat region */
5.     RegNumber++; /* Number Ri'l of this new flat region */
6.     Pix2cf[regElement] = RegNumber;
7.     push(pixS, regElement); /* init. of stack pixS to scan it*/
8.     while (pop(pixS, pixel) != 0)
9.       for all nghb of pixel
10.        if ((F[nghb] == L) && !Pix2cf[nghb]) /* arrow: cont. line */
11.          Pix2cf[nghb] = RegNumber;
12.          push(pixS, nghb);
13.        else if (F[nghb] > L) /* arrow: dots + dashes */
14.          node = Pix2cf[nghb]; /* node containing nghb and its */
15.          anc = Node2ances[F[nghb]][node]; /* ances. at L+1 */
16.          push(Fusion[anc], RegNumber);
17.          push(Fusion[RegNumber], anc);

```

Fig. 4. Procedure to detect the flat regions and to establish the inter-region connexions (step B1 L)

Variables As a general convention we use capitals at the beginning of a variable name only if the variable appears also outside the procedure. These variables are **F**, **Tree**, and **NodeNumb** as described above, **RegNumber** ($= R_{i'l}$ and = number of confiners at level $l + 1$ plus flat regions at l , after (B1 l)), $L = l$, and $R = r$, and the four data structures **Pix2cf**, **Fusion**, **Node2ances**, and **Region2conf**. These four variables, described already above and illustrated in Figure 3, are internal variables of (B) but **Pix2cf** and **Node2ances** are input and output for each step (B l) whereas **Fusion** and **Region2conf** are cleared after step (B2 l). **RegNumber** reappears indirectly in line 12 of (B2 l) but is an internal variable of (B l). Outside the procedures, after (A), **Pix2cf** is initialized to zero for each pixel, and we set **NodeNumb**[$R+1$]=0. All variables beginning with a small letter are internal variables of the respective procedures.

```

1. nodeNumber = 0; init(Region2conf); /*init. all array el. to 0*/
2. for each prevNode at level L+1 do
3.   if (Region2conf[prevNode] == 0)          /* new confiner at L*/
4.     nodeNumber++;
5.     Region2conf[prevNode] = nodeNumber;
6.     push(regS, prevNode); /* regS: stack of regions to fuse */
7.     while (pop(regS, reg1) != 0)
8.       while (pop(Fusion[reg1], reg2) != 0)
9.         if (Region2conf[reg2] == 0)
10.          Region2conf[reg2] = nodeNumber;
11.          push(regS, reg2);
12. for each flatReg at level L do
13.   if (Region2conf[flatReg] == 0) /* new leaf in the tree */
14.     nodeNumber++;
15.     Region2conf[flatReg] = nodeNumber;
16. NodeNumb[L] = nodeNumber;
17. for each regElement at level L do
18.   Pix2cf[regElement] = Region2conf[Pix2cf[regElement]];

```

Fig. 5. Procedure to associate the flat regions and the confiners at level $L+1$ with the new confiners (nodes). The association is kept in the array **Region2conf** (step B2 l , $l=L$)

Sub-steps (B1 l) and (B2 l) In the procedure in Figure 4, the flat regions are represented by the current value of **RegNumber** ($= R_{i'l}$). They are initialized in line 5-7 with **regElement** which corresponds in the example in Figure 3(a) to the encircled pixel in the middle at level 9. Pixels at the border are supposed to be set at 0 (see lowest row in Figure 3(a)). The procedure (Figure 4) detects each pixel in each flat region following a recursive scheme. It implements this scheme iteratively using the data stack **pixS**. We will remark here that we do not need a priority queue as in [7], where the pixels in the queue had to be ordered

according to their grey level. This is because all pixels which put in the queue have the same grey level L . The auxiliary function `push(pixS, element)` places `element` at the top of the stack `pixS` as `push(Fusion[number1], number2)` places the number `number2` at the top of the stack `Fusion[number1]`. `pop(pixS, element)` removes the element at the top of the stack and stores it in `element`. It returns 1 if there was an element in the stack, else 0. For a `pixel` fetched from `pixS` (line 8) we will examine all neighbors `ngnb` (line 9): (i) if the neighbor is in the same flat region and not treated up to now (this corresponds to the continuous arrows in Figure 3(a)), it will be treated in lines 11, 12; (ii) if the neighbor is in a confiner at a previous level (arrow with dashes and dots), lines 14-17 are executed; (iii) else, if the neighbor is at a lower level or already updated (dashes), nothing is done. In case (ii) the connexions between the lowest updated confiner containing the neighbor (represented by `anc`) and the current flat region (represented by `RegNumber`) are stored in `Fusion` (line 16, 17; Figure 3(b)).

The array `Region2conf` in Figure 5 is of length `RegNumber` and its indices correspond to the confiners/nodes at the previous level $l + 1$ followed by the flat regions at level l as represented by $R_{i'l}$ (Figure 3(b), line 2 and 5 in Figure 4). The $R_{i'l}$ are stored in the `Fusion` stacks. The procedure in Figure 5 implements step (B2 l) following a very similar scheme up to line 11 as that implementing (B1 l). The differences are that only the number of the new confiner has to be kept (line 10 which corresponds to line 11 in Figure 4 and lines 9-11 against lines 10-17 in Figure 4) and that the number of connexions changes for different regions and confiners (line 8 against line 9 in Figure 4). Lines 12-15 detect the new leaves. Figure 6 implements step (B4 l) as already described above. The remaining step (B3 l) is detailed in section 3.3.

Ways to Speed Up the Algorithm To improve in practice the calculation time, first, we will implement a way to store a connexion between a confiner and a flat region just once in the corresponding stack of `Fusion` during step (B1 l) (Figure 4). We use an array which marks for each confiner at the previous level if the current flat region has already been treated. We refer to this in section 5 as “`mark_neighConf`”. We achieve in a similar way, to deal with a tree implementation using unique representation of identical confiners where we mark the confiners adjacent to a flat region at the current level.

The second improvement for which we will present timings in section 5 is an attempt to implement all stacks in the `Fusion` array in one big array. It will be referred below as “`arrayFusion_Proc`”.

3.3 Propagating Information through the Tree

Let be given a rooted tree T known from its highest level down to a level l where the levels l decrease iteratively from the highest to the lowest one. This subsection addresses the *problem* of maintaining a data structure associated to T permitting, at each iteration, to find in constant time for any node at any level $l' > l$ its ancestor at level l .

```

init(Tree[L]); /* initial. Tree[L][node] to 0 for all nodes at L */
for each prevNode at level L+1 do
    node = Region2conf[prevNode];
    addNode(Tree[L][node], Tree[L+1][prevNode]);
for each regElement at level L do
    addPixel(Tree[L][Pix2cf[regElement]], regElement);

```

Fig. 6. Procedure updating the tree with the attributes of the confiners at level L (step B4 L)

In a first approach, we implement this data structure using the array `Node2ances` (see Figures 3(c), 4) associating a node to its ancestor at the required level l . We update `Node2ances` for each new level $l - 1$ by replacing for each already updated node the number of its ancestor at level l by that at level $l - 1$. This is illustrated in Figure 3(c). We call this procedure “`all_nodes_Proc`”.

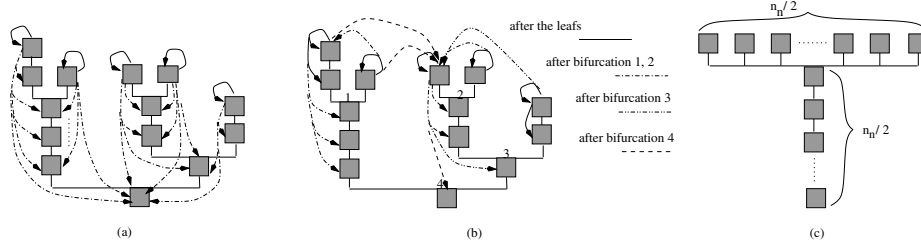


Fig. 7. (a), (b) Two ways to maintain a data structure permitting to find for each leaf in constant time its ancestor at level l ; (c) example for worst-case analysis of `all_leaves_Proc`

We can reduce the operations by associating to each node one of its descendant leaves. Then, we can reduce the above stated problem to leaves instead of nodes by implementing the required data structure as an array `Leaves2ances` and proceeding as in the procedure before, but updating the ancestor associations only for each leaf. This is illustrated in Figure 7(a). We call this procedure “`all_leaves_Proc`”. However, as we will see in section 4 the worst-case complexity of `all_leaves_Proc` stays $O(n^2)$ as for the procedure `all_nodes_Proc`.

We show now, how to stay – as proven in section 4 – within a worst-case complexity of $O(n \log n)$. We have to proceed as illustrated in Figure 7(b). For a given subtree, only for one leaf m we set an association to the subtree root. We can implement this association either directly as a new field of a leaf or as a separate list or array `SubtreeL2ances`. The other leaves in the subtree point to m and get the ancestor value from m . If a bifurcation appears in a node b , we have to change the pointers of the leaves in the new subtree rooted in b to point

all to one single leaf of this subtree. The number of operations for this step is limited by the number of leaves in the subtree (see the proof of proposition 2 in section 4, where we call this procedure “`subtreeLProc`”).

4 Complexity Analysis

Lemma 1. *If n is the number of pixels of an image and if n_n denotes the number of nodes in the corresponding confinement tree, we have $n_n \leq n$.*

Proof The lemma follows from the unique representation of identical confiners.

Given a rooted tree with n_n nodes (leaves included), how many nodes have to be passed through in order to go from all leaves to the root? One could think, the answer is $O(n_n \log n_n)$ as in two extremal cases – the complete binary tree and the tree with $n_n - 1$ leaves – at most $O(n_n \log n_n)$ passes are necessary, even only $O(n_n)$ in the latter case. But Figure 7(c) illustrates a case¹ where $O(n_n^2)$ passes are required. This is also the worst-case as there are at most $n_n - 1$ leaves and at most n_n nodes on the path between a leaf and the root. This proves the

Proposition 1. *The procedure `all_leaves_Proc`, given in section 3.3, updates the array `Leaves2ances` for all levels in a total time $O(n_n^2)$.*

Proposition 2. *The procedure `subtreeLProc`, given in section 3.3, updates the array `SubtreeL2ances` for all levels in a total time $O(n_n \log n_n)$.*

Proof We will prove that we have at most $n_n \log n_n$ operations without counting the $O(n_n)$ initialization operations. This, we will prove by induction on the number of nodes of the tree. The following logarithms are at base 2. For $n_n = 1$ (initialization of the induction) there is nothing to show. We will consider the root of the tree and suppose that it has k ($k \geq 2$; if $k = 1$ we have $(n_n - 1) \log(n_n - 1) + 1 \leq n_n \log n_n$) sons defining k subtrees for which the induction hypothesis is true. Therefore, and as we need less than n_n operations when passing from one level to the next one (see section 3.3), we can reduce the problem to the following statement: $\sum_{i=1}^k n_i \log n_i + n_n \leq n_n \log n_n$, $n_n = 1 + \sum_{i=1}^k n_i$, $n_i \geq 1$. As we are in the convex part of the function $x \log x$ (as $n_i \geq 1$) this follows directly from Jensen’s inequality for convex functions: we have $\frac{1}{k} \sum_{i=1}^k n_i \log n_i \leq (\frac{1}{k} \sum_{i=1}^k n_i) \log (\frac{1}{k} \sum_{i=1}^k n_i)$ from which follows $\sum_{i=1}^k n_i \log n_i + n_n \log k \leq n_n \log(n_n - 1) \leq n_n \log n_n$. As $1 \leq \log k$ ($k \geq 2$) this proves our statement and therefore our proposition 2.

Theorem 1. *The algorithm calculates the confinement tree in $O(n \log n)$ time using $O(n)$ scanning, fusion, attribute and $O(n_n \log n_n)$ propagation operations.*

¹ An image corresponding to this tree is possible, even for $n_n = n/2$

Proof We will just prove that the total time to execute step (B2 l) for all levels l is $O(n)$. The rest follows either from lemma 1, proposition 2 or is clear. In step (B2 l), for a given level l , each flat region number at l and each number of a confiner at level $l+1$ is put exactly once in the data stack **regS**. For each element in the data stack the number of operations is the number of its connexions with flat regions or confiners. The number of all this connexions at a level l is limited by 4 times the number of pixels (forming the flat regions) at grey level l as each pixel has at most 4 neighbors. Therefore, the total number of connexions for all levels l is within $O(n)$ and our theorem is proven.

5 Experimental Evaluation

We measured the execution times for the algorithm and its parts as presented in the procedures of the Figures in section 3.2 for several images on a Intel Pentium III, 450 MHz. The timings are measured with the `clock()`-function under LINUX operation system and the results are summarized in figure 8 as well as information about the images (they have all 8 bit grey values; the MRI brain image is presented and treated in [12,11] and the microscopic images A, B in [11]). We observe stable behavior of our algorithm for approximately the same number of leaves and nodes but also increasing importance of step (B3 l) for larger images with larger trees: the propagation operations are about 10 times more time consuming for the microscopic images C and D with respect to A and B, whereas the other operations increase only 3-4 times. However, as $n_n \ll n$ the propagation operations does not become dominant. At the contrary, Figure 2(b) illustrates that the execution time of the algorithm of Jones [8] can become prohibitive *even if* $n_n \ll n$: for an increasing number of maxima (5 in 2(b)), all operations are multiplied by a factor depending linearly on this number. Figure 2(b) presents a synthetical image, however, the timings measured by Jones [8] on real images allow to conclude that our algorithm can be 10 up to 50 times faster in practice.

6 Conclusion

We presented a new algorithm for calculating the confinement tree and could prove that its worst-case complexity is $O(n \log n)$. Moreover, the algorithm separates different kinds of operations such that all operations except propagation stay in $O(n)$ time. We showed how to achieve complexity $O(n_n \log n_n)$ for propagation operations where n_n is the number of nodes in the tree (in a first approach, we implement propagation in $O(n_n^2)$, sufficient if $n_n \ll n$). Two example images illustrate $O(n^2 \log n)$ complexity for a known algorithm, prohibitive computation time for this known algorithm even if $n_n \ll n$, and $O(n^2)$ for an algorithm calculating a tree related to the confinement tree. Practical evaluation allows to conclude that our improvement leads to much faster calculation times on standard medical images.

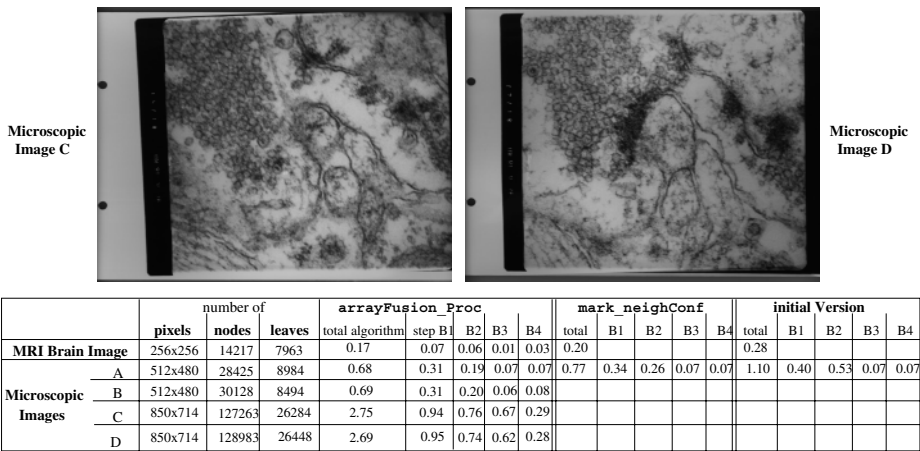


Fig. 8. Test images and execution times in seconds for the implementation of our algorithm

Acknowledgments

We would like to thank Prof. M. Coupri  for helpful bibliographic remarks.

References

1. Chassery, J.-M., Montanvert, A.: G om trie discr te en analyse d'images. Hermes (1991) 395

2. Cormen, T. H., Leiserson, L. E., Rivest, R. L.: Introduction to algorithms. MIT Press (1998) 396

3. Coupri , M., Bertrand, G.: Topological grey scale watershed transformation. In: SPIE Vision Geometry V Proceedings, Vol. 3168 Bellingham, WA (1997) 136–146 392, 393

4. Guillataud, P.: Contribution   l'analyse dendronique des images. PhD thesis, Universit  de Bordeaux I (1992) 392, 393

5. Hanusse, P., Guillataud, P.: S mantique des images par analyse dendronique. In: AFCET, 8th RFIA, Vol. 2. Lyon (1992) 577–588 392, 393, 394

6. Hartigan, J. A.: Statistical theory in clustering. J. of Classification 2 (1985) 63–76 392, 394

7. Jones, R.: Connected Filtering and Segmentation Using Component Trees. Computer Vision and Image Understanding 75 (1999) 215–228 392, 393, 394, 395, 399

8. Jones, R.: Connected Filtering and Segmentation Using Component Trees: Efficient Implementation Algorithms. <http://www.dms.CSIRO.AU/~ronaldj/pseudocode> (1999) 393, 394, 395, 403

9. Kok-Wiles, S.L, Brady, J. M., Highnam, R.: Comparing mammogram pairs for the detection of lesions. In: Karssemeijer, N. (ed.): 4th Int. Workshop of Digital Mammography, Nijmegen, Netherlands, June 1998. Kluwer,Amsterdam, (1998) 392

10. Mattes, J., Demongeot, J.: Dynamic confinement, classification, and imaging. In: 22nd Ann. Conf. GfKI, Dresden, Germany, March 1998. Studies in Classification, Data Analysis, and Knowledge Organization. Springer-Verlag (1999) 205–214 [392](#), [394](#)
11. Mattes, J., Demongeot, J.: Tree representation and implicit tree matching for a coarse to fine image matching algorithm. In: MICCAI'99, C. Taylor, A. Clochester (Eds.). LNCS. Springer-Verlag (1999) 646–655 [392](#), [393](#), [394](#), [403](#)
12. Mattes, J., Richard, M., Demongeot, J.: Tree representation for image matching and object recognition. In: DGCI'99, G. Bertrand and M. Couprié and L. Perroton (Eds.). LNCS. Springer-Verlag (1999) 298–309 [392](#), [403](#)
13. Salembier, P., Oliveras, A., Garrido, L.: Antiextensive Connected Operators for Image and Sequence Processing. IEEE Trans. on Image Processing **7** (1998) 555–570 [392](#), [393](#), [394](#), [395](#)
14. Wishart, D.: Mode analysis: A generalization of the nearest neighbor which reduces chaining effects. In: Cole, A. J. (Ed.): Numerical Taxonomy. Academic Press, London (1969) 282–319 [392](#), [394](#)
15. Zuck, P., Lao, Z., Skwish, S., Glickman, J. F., Yang, K., Burbaum, J., Inglese, J.: Ligand-receptor binding measured by laser-scanning imaging. PNAS **96** (1999) 11122–7 [393](#)