COMPONENT TREES FOR IMAGE FILTERING AND SEGMENTATION

Ronald Jones

CSIRO, Mathematical and Information Sciences,
Locked Bag 17, Nth Ryde N.S.W. 2113, AUSTRALIA,
Email: ronald.jones@cmis.csiro.au

ABSTRACT

In this paper we present algorithms for non-flat connected component filters using the notion of a component tree. The advantage of using non-flat filters as compared to flat filters is that they can access and utilise linking between components at sequential gray-levels in the image. We have found that this information can be used to develop powerful new connected filters with practical applications [1]. One of the key benefits of the approach is that the image features to be filtered undergo the maximum amount of filtering that is possible while leaving the rest of the image untouched. As a consequence, a segmentation of the features can then be obtained simply by locating those pixels within the image that have been changed by the filter.

1. INTRODUCTION

In binary morphology, a component filter preserves only those connected components in the image that satisfy a given criterion [2]; the remaining components are removed. The criterion is based on one or more of the component's attributes, for example the area or perimeter of the component. Any binary filter can be used to construct a corresponding gray-level filter by using the notion of *threshold decomposition.* A component in a gray-level image is defined as a connected set of pixels in a threshold set of the image. The disadvantage with flat filters is that they cannot access and utilise the link between components at sequential gray-levels in the image, as by construction they are applied to each threshold set separately. In this paper we generalise work presented in [3] by introducing a new gray-level component filter that is not-necessarily flat, using the notion of a *component tree* [1].

2. THE COMPONENT TREE

The component tree is a representation of a gray-level image that contains information about each image component and the links that exist between components at sequential gray-levels in the image. We define a component tree as a set of *nodes* connected by a set of *edges.* Each node in the tree represents a particular component in the gray-level image. The node is an abstract representation of the component that may be anything from a list of all the pixels in the component to a single attribute value such as component area. An example of a component tree is illustrated in Fig. 1b, corresponding to the small gray-level image shown in Fig. 1a. For every component in that image, there is a corresponding node in the tree, represented by the circles in the figure. The *root* of the tree is shown at the bottom of the figure and the *leaves* of the tree are indicated by the circles in bold at the top of the figure. A *branch* is defined as the shortest sequence of linked nodes from any given leaf down to the root of the tree. There are three branches in the tree, corresponding to the three leaves. The lines drawn between the nodes are the edges and show the links between components in the image. As an example of the attribute values that can be stored in the nodes, component area is shown in italics to the right of each node.

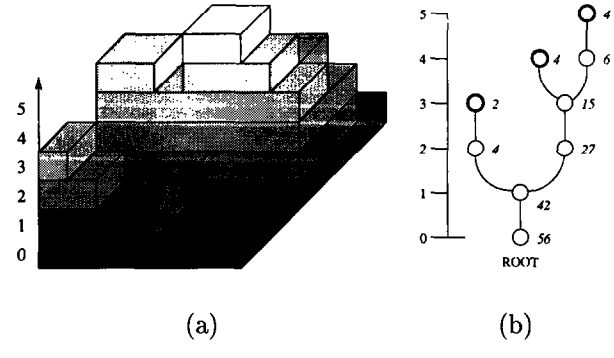

Figure 1: The component tree. (a) Perspective view of an example image. (b) Corresponding component tree.

3. FILTERING AND SEGMENTING AN IMAGE WITH A COMPONENT TREE

The next step is to introduce mechanisms to filter the image using the component tree. Filtering the tree is a decision process which classifies nodes into those that are to be removed and those that are to be preserved. The component tree contains both attribute and linking information and so there are many different possible tree filters that can be constructed. Of particular interest to us is one which which is based on the concept of an *attribute signature*. An attribute signature is simply the sequence of node attributes in a branch of the component tree. Necessarily then, there is an attribute signature for every leaf in the tree. Tree filtering using attribute signatures is a decision-making process which classifies leaves into those that are active and those that are not, based on the leaf's attribute signature. The entire branch must be classified as active when the leaf is active. One of the key benefits of the approach is that the image features to be filtered undergo the maximum amount of filtering that is possible without changing the rest of the image at all. This is accomplished by finding leaves that lie outside the region to be filtered and classifying each corresponding branch as active. In so doing, we ensure that these regions cannot be affected at all by the filtering process. On the other hand, those leaves that inside the region to be filtered remain classified as not active. This ensures that these regions are filtered as much as possible.

The concept is demonstrated on the image shown in Fig. 2a, where the filtering problem is to remove as much of the background as possible while preserving the face. One method is to combine an area signature with an eccentricity signature, as the face is elliptical in shape. The attribute signature for a typical regional maximum within the the face is shown in Fig. 3a. The attribute used is area; the values in the signature fall monotonically from the area of the image down to the value of 1, which is the area of the regional maximum used. Shown in Fig. 3b is the attribute signature for the same regional maximum, this time using component eccentricity as an attribute. By inspection of the two signatures, a suitable criterion can be devised; the result from the filter is shown in Fig. 2b. Notice that the face has been preserved completely, while the background has been removed as much as possible without affecting the components that are connected to the face. As mentioned above, this is a key feature of image filtering using attribute signatures.

The component tree is used for image segmentation as well as image filtering. The approach is to consider a segmented image to be simply a binary image which

(a) (b)

Figure 2: Filtering an image using attribute signatures.

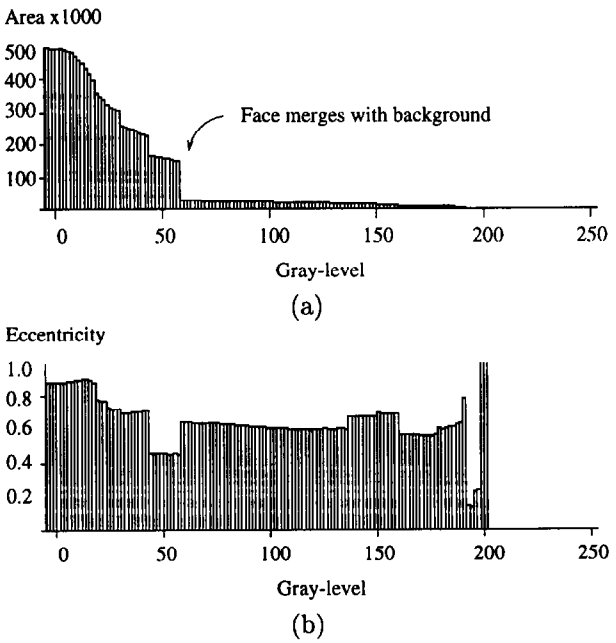

Figure 3: Attribute signatures for the image of the Mona Lisa.

marks only those pixels in the input image that have been changed by the gray-level component filter. This approach lends itself well to filtering based on attribute signatures, where objects to be filtered undergo the maximum possible amount of filtering while the rest of the image is unchanged. In this case, the segmented image marks only those pixels within the objects that have been filtered. An example of a segmentation is shown in Fig. 4. This time the face has been filtered, rather than the background as shown in Fig. 2b. The resulting segmentation of the face marks those pixels that have been changed by the filter.

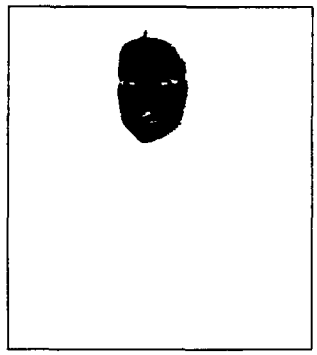

Figure 4: Segmenting an image using attribute signatures.

4. EFFICIENT IMPLEMENTATION ALGORITHMS

In this section we present pseudo-code for efficient algorithms to implement image filtering and segmentation using component trees. The process of filtering and segmenting an image using a component tree can be been divided into the three stages: (1) Constructing the tree from the image; (2) Filtering the tree (specifically, using an attribute signature); (3) Mapping the filtered tree to an output image. These three steps are illustrated in Fig. 5. Modularising the process in such a way allows a flexible and efficient tree-based image processing package to be developed. Each node in the tree corresponds to a component in the gray-level image; filtering the tree involves deciding which components are to be removed from the image during filtering. The mechanism used is the binary flag *active* in the information structure associated with each tree node. If this flag is set to the value 1, then the component corresponding to that node is to be preserved. If it is set to 0, then the component is to be removed. This concept is illustrated in Fig. 5, where the nodes that have been set to active during stage (2) are indicated by the filled-in circles. The image resulting from stage (3) contains only those components that have corresponding active nodes in the filtered tree.

In Fig. 6 is shown the tree data structures that will be used. Shown in Fig. 6a is the *root* data structure, containing a pointer *root* to the root node in the tree, the number *nleaves* of leaves in the tree and a structure *leaves* containing pointers to all the nodes in the tree that are leaves (this facilitates convenient traversal of the tree from a leaf to the root). Shown in Fig. 6b is the *node* data structure, containing a pointer *info* to the information associated with the node, a pointer *parent* to the parent node of the node, the number *nchildren* of

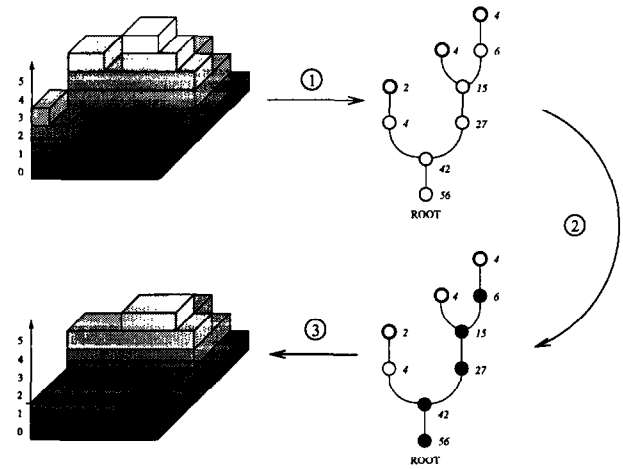

Figure 5: The three stages of filtering and segmenting an image using a component tree.

children of the node and a structure *children* containing pointers to all the children. The *parent* pointer allows traversal of the tree from a leaf to the root, while the *children* pointers allow traversal from the root to the leaves. In Fig. 6c is shown the information data structure *info*, containing: (i) The gray-level of the node, *graylevel*; (ii) The location of the node, where only one location within the component need be recorded (the rest of the component can be obtained by scanning for pixels connected to this point with a value greater than or equal to *graylevel*); (iii) The attribute assigned to the node; and (iv) A flag *active* to be set during the tree filtering stage.

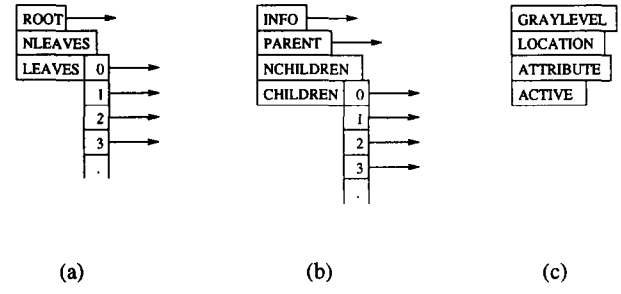

Figure 6: Data structures used for the tree. (a) The *root* data structure. (b) The *node* data structure. (c) The *info* data structure.

4.1. Constructing the Component Tree

The following algorithm can be used to construct a component tree from a gray-level image. It is based on the algorithm for attribute openings and thinnings presented in [3], which in turn was a generalisation of algorithm presented by Vincent [2] for area openings. The algorithm requires the following input variables: **f**, a copy of the input image with the border pixels set to zero (to stop the algorithm running off the side of the image). **size**, the number of pixels in f; **min**, the minimum gray-level in f; **RegA**, an array containing the locations and gray-levels of all the regional maxima in the image; **numreg**, the number of regional maxima in the image. The algorithm outputs the variable **tree** with a structure as discussed above. In addition, the algorithm also utilises the following auxiliary functions: **push**(Q,*pixel*), places *pixel* on the priority queue Q; **pop**(Q,*pixel*), fetches the *pixel* in the Q with highest gray-level. It returns 0 if the queue is empty and 1 if it finds a pixel; **update**(*pixel*), updates attribute information about the current component being scanned by including *pixel* in the current component; **clear**(), clears any information collected by *update*; **Nbr**(*oldloc*), returns the neighbourhood locations around the given location *oldloc*; **create_node**(*level*, *location*), allocates memory for a new node and assigns it the given gray-level, image location and the current attribute information collected by *update*; **attach_node**(*newnode*, *node*), makes *node* a child of *newnode*; and **attach_at_level**(*leaf*, *node*, *level*), attaches *node* to the branch containing *leaf* at the gray-level *level*.

construct_tree(f,size,min,RegA,numreg)

```
1. tree.root ← create_leaf(min, size/2)
2. tree.nleaves ← numreg; regindex ← 0
3. for all pixel ∈ RegA
4.     clear()
5.     regindex ← regindex + 1; level ← pixel.level;
           location ← pixel.x
6.     newl[pixel.x] ← regindex; node ← NULL;
           flag ← 0
7.     push(Q, pixel)
8.     while (pop(Q, pixel) ≠ 0)
9.         if (pixel.level < level)
10.            newnode ← create_node(level, location)
11.            if (node = NULL)
12.                tree.leaves[regindex − 1] ← newnode
13.            else
14.                attach_node(newnode, node)
15.            node ← newnode; level ← pixel.level
16.            if (level = f2[pixel.x])
17.                attach_at_level(tree.leaves[oldl[pixel.x]
                       −1], node, level)
18.            while (pop(Q, pixel) ≠ 0)
19.                .
20.            flag ← 1; break
21.        oldloc ← pixel.x; oldl[oldloc] ← regindex;
               f2[oldloc] ← level
22.        update(pixel)
23.        for all pixel.x ∈ Nbr(oldloc)
24.            if ((f[pixel.x] ≠ min) and (newl[pixel.x]
                       < regindex))
25.                pixel.level ← f[pixel.x]; newl[pixel.x]
                       ← regindex
26.                push(Q, pixel)
27.    if (flag = 0)
28.        newnode ← create_node(level, location)
29.        if (node = NULL)
30.            tree.leaves[regindex − 1] ← newnode
31.        else
32.            attach_node(newnode, node)
33.        node ← newnode
34.        attach_node(tree.root, node)
35.return tree
```

4.2. Filtering the Component Tree

Each node in the tree corresponds to a component in the gray-level image; filtering the tree involves deciding which components are to be removed from the image during filtering. The mechanism used is the binary flag *active* in the information structure associated with each tree node. If this flag is set to the value 1, then the component corresponding to that node is to be preserved. If it is set to 0, then the component is to be removed; it is assumed that this flag initially has the value 0. The tree can then be passed onto the image filtering and segmentation stage presented in Section 4.3. In this section we present pseudo code for a filter based on attribute signatures.

The following algorithm *signature_filter* can be used in conjunction with a user-defined function *test_signature* to filter a tree using attribute signatures. The function *test_signature* must be designed for the particular image filtering problem at hand. It should return 0 if the signature belongs to a regional maximum within the feature to be filtered or segmented, otherwise it should return a non-zero value. The algorithm requires a single input variable **tree**.

signature_filter(tree)

```
1. i ← 0
2. while (i ≠ tree.nleaves)
```

```
3.    if (test_signature(tree.leaves[i]) ≠ 0)
4.       set_actives(tree.leaves[i])
5.    i ← i + 1
```

The simple recursive function *set_actives* defined below. By setting the flags of the entire branch to the value 1, we ensure that this regional maximum will not be altered by the subsequent image filtering and segmentation phase. The input variable to *set_actives* is **node**, which is a leaf in the tree (cf. line 4 of the above algorithm).

set_actives(node)

```
1. if (node.info.active = 0)
2.    node.info.active ← 1
3.    if (node.parent ≠ NULL)
4.       set_actives(node.parent)
```

The execution time for the algorithm *signature_filter* using an input tree with 10 thousand leaves is approximately 1.5 seconds on a Sparc 10.

4.3. Algorithm for Filtering and Segmenting the Image

Once the tree has been filtered, the following in-place recursive algorithm can be used to filter and segment the corresponding image. The input variables are: **f**, the memory allocated for the gray-level filtered image, initialised to be a copy of the input image with the border pixels set to zero; **s**, the memory allocated for the binary segmented image, initialised to the value zero; and **node**, initialised to be the root of the filtered tree.

filter_image(f,s,node)

```
1. i ← 0
2. while (i ≠node.nchildren)
3.    if (node.children[i].info.active = 1)
4.       filter_image(f, s, node.children[i])
5.    else
6.       flatten(f, s, node.children[i].info.location,
            node.info.graylevel)
7.    i ← i + 1
```

The subroutine *flatten*, defined below, utilises the following variables and auxiliary functions: **f** and **s**, images as defined above; **x**, **y** and **z**, locations within the image; **level**: gray-level threshold - all pixels in the region connected to **x** with value greater than *level* are flattened; **Q**, a simple queue containing the locations of pixels in the image; **push**, a function taking two arguments, the first being the queue Q and the second the location x to be to be stored in the queue. The function places the location x as the first item in the queue; **pop**, a function which takes the same two arguments as **push**. It fetches the last item found in the queue Q, returning 1 if it finds an item and 0 if the queue is empty; and **Nbr**, the neighbourhood function, as defined in Section 4.1.

flatten(f,s,x,level)

```
1. if (f(x) > level)
2.    push(Q, x)
3.    f(x) ← level; s(x) ← 1
4. while (pop(Q, y) ≠ 0)
5.    for all z ∈ Nbr(y)
6.       if (f(z) > level)
7.          push(Q, z)
8.          f(z) ← level; s(z) ← 1
```

For a 1000×1000 image with just over 10 thousand regional maxima, the tree requires 3 mega-bytes of disk space. The total implementation time for the three stages for processing an image using a component tree is of the order of 50 seconds on a Sparc 10 (Model 51, 50MHz).

5. CONCLUSION

This paper has introduced a general non-flat gray-level connected filter and proposed efficient algorithms for its implementation. One of the key benefits of the approach is that the image features to be filtered undergo the maximum amount of filtering that is possible while leaving the rest of the image untouched.

6. REFERENCES

[1] R. Jones. Connecting filtering and segmentation using component trees. Submitted to Computer Vision and Image Understanding, May 1997.

[2] L. Vincent. Grayscale area openings and closings, their efficient implementation and applications. In J. Serra and P. Salembier, editors, *Mathematical Morphology and its applications to signal processing*, pages 22–27. UPC Publications, May 1993.

[3] E. J. Breen and R. Jones. Attribute openings, thinnings and granulometries. *Computer Vision and Image Understanding*, 64(3):377–389, November 1996.