Component tree: an efficient representation of grayscale connected components

Gaëtan Lehmann[1]

December 31, 2007

[1]INRA, UMR 1198; ENVA; CNRS, FRE 2857, Biologie du Développement et Reproduction, Jouy en Josas, F-78350, France.

Abstract

Connected component is a well known and very useful notion in binary case. As often in mathematical morphology, this notion can be extended to the grayscale case, and allow to perform lot of the useful transforms based on binary connected components in the grayscale images. This article describe the *component tree*, a data structure able to efficiently represent the grayscale connected components in an grayscale image, as well as the algorithm used to build the component tree.

All the source codes are provided, as well as a full set of tests and several usage examples of the new classes.

Contents

1 A few definitions and properties **1**

2 Component tree **2**

2.1 Data structure . . . 2

2.2 Implementation . . . 3

2.3 Comparison with implementations described in references . . . 4

2.4 Comparison with the binary implementation . . . 5

Line representation vs linked list . . . 5

Index vs offset . . . 6

Support for large number of attributes vs single templated attribute . . . 6

Parallel vs recursive attribute computation . . . 6

2.5 Computation of the component tree . . . 6

1 A few definitions and properties

The connectivity defines which pixels are in the neighbourhood of a pixel. The same connectivity is usually defined for all the pixels in an image.

A binary connected component is defined as a set of foreground pixels for which at least one path exists between all the pairs of pixels of the connected component, in the graph formed by the pixels of the image and the connectivity used for that image.

The connected component notion can be extended to grayscale images with the notion of *threshold set*. A threshold set is the set of foreground pixels formed by selecting all the pixels greater or equal to a threshold value. The connected components can be found at a defined threshold: that's the (binary) connected components in the threshold set. As a consequence, there are as much connected components sets as pixel values in a grayscale image. Also, it can be noticed that a connected component at a given threshold can't partially contain a connected component at a higher threshold – the connected component is fully included or is not included. This is a very interesting property of grayscale connected components, because it make the transforms based on that notion never add or move any frontier in the image – they can only be removed. That property makes possible to represent the grayscale connected components as a tree: the component tree.

The above definition of the threshold set is valid for the images where the objects are bright and the background is dark. If the image contain dark objects, the pixels must be selected if they have a value lower or equal to the threshold value.

In the various articles about connected component, the term *max-tree* is often used in place of component tree. In that article, the term *max-tree* is used for the component trees which are representing bright objects, and *min-tree* for the component trees which are representing dark objects. The term *component tree* is used when the notion of bright or dark object is not important.

In some articles, *max-tree* and *component tree* are used to define two variants of the same data structure – it's not the case in that article.

2 Component tree

The component tree is an efficient representation of all the connected component sets in a grayscale image.

2.1 Data structure

The component tree is a tree, and thus contains nodes which have:

- a parent node,
- some child nodes.

The *root* node is the only node with no parent. A node without any child is called a *leaf*. Each node represent a connected component at a given threshold, and is thus associated with:

- the pixel value used as threshold,
- the list of pixels in that connected component.

However, each node does not list all the pixels in the connected component, but only the ones with the exact pixel value associated with that node. Thanks to the inclusion relation with the connected components at higher threshold values, all the pixels of the connected component can be computed as the union of the pixels in the node, and the pixels of all its children.

Finally, a node is often associated to an *attribute*, often a value of a specific property of the connected component (like its size, its shape, its intensity, ...), or the the relation with the other connected components of the tree (like the intensity variation with the parent node, the number of children, ...). In practice, many kind of attributes can be associated to a node.

2.2 Implementation

A particular attention has been ported both on usability and on performance of the code. For usability, and following the approach used in ITK, the code has been implemented using the object paradigm style. However, a much rough approach has also been used in some case, because of the significant performance improvements.

There are two critical points for usability, and for performances:

- the nodes,
- the indexes.

Both of them may be found in important number in the component tree. However, it must be noticed that:

- there are less nodes than indexes,
- the nodes are much often manipulated by the developers than the indexes,
- the number of indexes is known, and always the same in an image, while the number of nodes is unknown before building the tree, and can change with the transforms,
- for both indexes and nodes, the order in the children and in the indexes containers is not relevant,
- while merging nodes, both indexes and nodes (children) containers must be merged as well, and so the merging must be efficiently implemented (if possible, in constant time) in the indexes and nodes containers,
- the nodes is a set of different kind of data, while the indexes are simply a position in an image.

Given the usability and performances considerations given above, it has been chosen to implement:

- the indexes as a simple offset in the image, of type $long$. It can be easily converted to a full *itk::Index* with the method provided in the *itk::ImageBase* class, and is much efficient both in memory and in computation time than the *itk::Index*. The memory usage for example is D times much efficient, where D is the image dimension.
- the index list as a custom list, which takes advantage of some particularities cited above (see below for much details).
- the nodes as a new class, *itk::ComponentTreeNode* (see below for much details).
- the children list as a *std::list*. This data structure can be efficiently merged (in constant time).

The proposed implementation is based on two main classes: *itk::ComponentTree* and *itk::ComponentTreeNode*.

itk::ComponentTreeNode is the class for all the node of a component tree. It contains several variables dedicated to that task:

- *Pixel* is the pixel value,
- *Parent* is a pointer to the parent node,
- *Children* is a list of the children nodes,
- *FirstIndex* and *LastIndex* are the first and last indexes of the index list (see below),
- *Attribute* is the attribute associated with the node.

The list of indexes is implemented in a compact way. It is stored in two places:

- An array of type *std::vector¡long¿* of the same size than the image is stored in the *itk::ComponentTree* class. The indices of the array correspond to the indexes in the image. Each content of that array is a reference to the next element in a list of indexes, or -1 if that the last element.
- The first index of the indexes list is stored in *itk::ComponentTreeNode*, as well as the last index, to allow a constant time merge.

This way of storing the indexes lists requires far less memory than with a *std::list*:

- $(N+2)L$ for the implemented way,
- $(3N+2)L$ with the *std::list*, because of the link to the previous and the next node,

where L is the size of the $long$ type on the system, and N the size of the list.

There can be a quite important number of nodes in a tree, so *itk::ComponentTreeNode* is a critical class for the performances of the transforms based on the component trees, and thus has not been implemented as a subclass of *itk::LightObject*, to avoid the cost of the smart pointers management, and the cost of the object factories.

2.3 Comparison with implementations described in references

The component tree implementation is described in several articles, sometimes in details, sometimes very roughly. In general, the tree is implemented as a *rooted tree*, where each element contains a pointer to its parent, or to the canonical element of its node. The rooted tree is an array of the same size than the image, where all the elements are a index to a another element in the same array. The attributes are stored in another structure outside, and the pixel value associated with the nodes can be retrieved from the original image.

This way of storing the component tree has some advantages compared to the proposed implementation:

- the memory usage is the smallest known one.
- attribute computation may be much efficient, by scanning the rooted tree in raster order.

- it allow direct access to the node from the pixel position.

but it also has some problems:

- the structure may not be canonical (all the element in the rooted tree points to the parent node, or to the canonical node)
- there is no direct link between the attribute and the node. It implies some extra work to keep the data structure used to store the tree and the data structure used to store the attribute values synchronised.
- if each element of the rooted tree is associated with an attribute value, storing those values is highly less memory efficient than storing an attribute value for each node.
- removing a node, by merging it in its parent, has a $O(n)$ complexity, where n is the number of pixels in the removed node, if the elements of the nodes are known, and $O(N)$, where N is the size of the image, if the elements are not known, which is the much common situation. The complexity is $O(C)$ with the proposed structure, where C is the number of children. Also, the rooted tree requires to modify the input image, or to recreate a new image similar to the input one when the tree is manipulated. With that last case, and depending on the pixel type, it can highly decrease the memory performance of the rooted tree implementation.
- determine if a node is a leaf or not is quite difficult. It can be done in constant time with the proposed implementation.

Note that some of the problems above are irrelevant with floating point pixel type, because, most of the time in that case, a node contains a single pixel.

2.4 Comparison with the binary implementation

The binary case has been implemented in another contribution to ITK. The differences are studied in that section.

Line representation vs linked list

The indexes contained in an object are implemented with the *run-length encoding* in the binary case. With that implementation, it is possible to store a high number of indexes in only two values:

- an index,
- a line length.

It is particularly efficient for the connected components, where lots of pixels are likely to be neighbours on a line. It also makes possible to optimise the computation of some attributes by avoiding the iteration over all the pixels contained in a line coded that way.

It would have be perfectly possible to use that encoding in the component trees. However, the component tree only store the indexes at a single pixel value – all the connected component is not store in each node. This data structure make less likely to have to code a line of at least two pixels. Also, the lists of indexes shown above would have more difficult to create with that encoding.

Index vs offset

In the binary case, the indexes used in the run-length encoding are stored as *itk::Index*. It is the more standard way to code an index in ITK. However, it requires N fold more memory than a simple offset in an image, where N is the dimension of the image. Because the component tree code doesn't use the run-length en coding, it uses a lot more indexes that in the binary case, so it has been chosen to use the offsets rather than the *itk::Index* to reduce the memory usage.

Support for large number of attributes vs single templated attribute

The binary object representation has been made to support several attributes at a time. The component tree node is made to support only a single templated attribute. There are two main reason for that difference:

- the binary object are often used to read the attribute values of an object, to describe the object for example. It is often interesting in that case to be able to access several attributes at a time. Reading the attribute values on a component tree node seem to be a very rare use case.
- the number of objects in the binary case is likely to be a lot smaller than the number of nodes in the component trees. Given the high number of nodes, it is safer to store a single (templated) value per node, and compute the value of another attribute if needed, rather than storing several attribute in a node. It is possible however to store several attributes, by using a *structure* as attribute type.

Parallel vs recursive attribute computation

In the binary case, all the objects are distinct. It is easy to compute the attribute values of several objects in parallel. In the component trees, the objects are not as well separated: a connected component is made of the pixels associated to a node, and of the pixels of all its children. Also, computing the attribute value of a node is often made from the values of its children. The attribute computation is naturally implemented recursively, and thus is much difficult to implement in parallel.

2.5 Computation of the component tree

An usual *itk::Image* can be converted to an *itk::ComponentTree* with the specialisations of *itk::ImageToComponentTreeFilter*: *itk::ImageToMaximumTreeFilter* for the bright objects and *itk::ImageToMinimumTreeFilter* for the dark objects.

The algorithm implemented is a slight modification of the algorithm from Najman and Couprie. It use the data structure described above to reduce the memory usage required by the algorithm: during the build of the component tree, the *Parent* is the last node known to be the current root of the tree.

References

[1] L. Ibanez and W. Schroeder. *The ITK Software Guide*. Kitware, Inc. ISBN 1-930934-10-6, http://www.itk.org/ItkSoftwareGuide.pdf, 2003.