Fast Implementation of the Ultimate Opening

Jonathan Fabrizio[1,2] and Beatriz Marcotegui[1]

[1] MINES Paristech, CMM- Centre de morphologie mathématique,
Mathématiques et Systèmes, 35 rue Saint Honoré - 77305 Fontainebleau cedex, France
[2] UPMC Univ Paris 06
Laboratoire d'informatique de Paris 6, 75016 Paris, France
{jonathan.fabrizio}@lip6.fr, marcoteg@cmm.ensmp.fr
http://www-poleia.lip6.fr/~fabrizioj/

Abstract. We present an efficient implementation of the ultimate attribute opening operator. In this implementation, the ultimate opening is computed by processing the image maxtree representation. To show the efficiency of this implementation, execution time is given for various images at different scales. A quasi-linear dependency with the number of pixels is observed. This new implementation makes the ultimate attribute opening usable in real time. Moreover, the use of the maxtree allows us to process specific zones of the image independently, with a negligible additional computation time.

1 Introduction

The ultimate opening (*U.O.*) is a powerful morphological operator that highlights the highest contrasted areas in an image without needing any parameters. It has been used, for example, as a segmentation tool for text extraction [1]. We offer, in this paper, a fast implementation of the *U.O.* based on a maxtree.

The article is organized as follows: in the first part, the *U.O.* operator is briefly explained. In the second part, the fast implementation is explained and an adaptation of the algorithm is made, to implement the iterated U.O., which we introduce. In the last part the speed of the algorithm is measured and then comes the conclusion.

2 The Ultimate Opening Operator

Introduced by Beucher [2], the *U.O.* is a residual operator that highlights patterns with the highest contrast. The operator successively applies increasing openings γ_i (of size i) and selects the maximum residue r_i computed between two successive results of opening, γ_i and γ_{i+1}, applied to the image (i.e. the difference $\gamma_i - \gamma_{i+1}$). The examples of Figs. 1 and 2, show that whatever small variations and noise, the operator only keeps the strongest patterns of the image: small variations on a contrasted structure are considered as noise and are automatically eliminated. For each pixel, the operator gives two pieces of information, the maximal residue ν and the size q of the opening leading to this residue:

M.H.F. Wilkinson and J.B.T.M. Roerdink (Eds.): ISMM 2009, LNCS 5720, pp. 272–281, 2009.

$$\nu = sup(r_i) = sup(\gamma_i - \gamma_{i+1}) \quad (1)$$

$$q = \max(i) + 1; r_i = \nu(\neq 0). \quad (2)$$

ν, called the transformation in the literature, gives indications on the contrast of pattern and q, usually called residual or associated function, gives an estimation of the pattern size (a sort of granulometric function). It is also possible to use an attribute opening [3]. In this case, the associated function provides information linked with the given attribute. In this article, all tests and comments have been made on ultimate attribute openings. The attribute we have chosen is the height of the connected component, defined as the maximum difference of vertical coordinate among pixels that belong to the connected component. We have chosen this attribute according to our application. Height attribute is easily computed during maxtree creation. Other attributes could have been used instead.

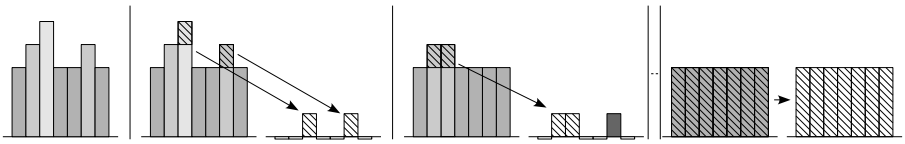

Fig. 1. The computation of the ultimate opening. From left to right: 1. the profile of the image is given, then 2. an opening of size 1 (γ_1) does not change the image and γ_2 removes two maxima. The residue is recorded in the transformation. 3. γ_3 generates a residue with the result of the previous opening, this residue is recorded in the transformation. Following openings (γ_4 to γ_7) do not generate residues. 4. At the end, γ_8 generates a bigger residue than the previous ones and is recorded in the transformation. As this residue is high, it erases all previous residues.

Fig. 2. On the left hand-side, an image, on the right hand-side the result (the transformation) of the ultimate height opening. Image copyright *institut géographique national (IGN).*

3 The Implementation Based on Maxtree

The *U.O.* operator is a very powerful non-parametric operator. Its direct implementation (by applying successive openings) leads to a high computation time. A faster implementation has been proposed by Retornaz [4,5] but it is still time consuming because many regions of the image are processed several times.

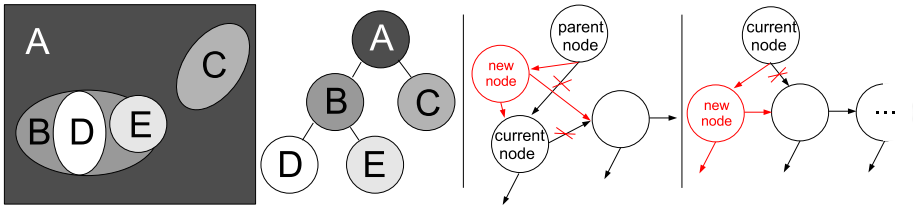

Fig. 3. From left to right: The image and the corresponding maxtree. The insertion of a node before in the branch (function *next_label_lower*()). The insertion of a node after in the branch (function *next_label_higher*()).

We propose a much faster implementation of the *U.O.* operator. This implementation is based on the use of *maxtree* [6] and has multiple advantages:

- the image is processed only twice: once to create the tree and the other to deduce the transformation and the associated function,
- a region may be processed independently from the rest of the image by processing the corresponding branch of the tree,
- once the maxtree is created, many other filters [7] can be applied to the image directly on the tree without processing the image (which is much faster [8]).

The maxtree is a data structure that represents the image by a tree (Fig. 3). A node of level l is linked with a subset of pixels at level l in the image. Those pixels must be connected together by pixels laying at a level higher or equal to l. This means that a node and all its sub-nodes represent a region of the image which is the union of connected regions of level equal or higher than l. Two nodes at the same level l are separated in the image by pixels at a level lower than l. The mintree is the dual structure. Our algorithm is divided into three parts: 1. building the maxtree according to the image, 2. processing the tree and 3. generating the output of the operator. We will study each step in details.

3.1 Building the Maxtree

Algorithms to compute the tree already exist [9,6,10] and [11]. Any of them could have been used. In order to have the description of all the process steps and because the creation of the tree is time consuming, the tree creation is revisited.

Before writing the algorithm, the first thing to do is to choose data structures. Huang et al. in [10] offer to use a combination of a linked list and *a hash map*. The linked list is used to chain the tree nodes and the hash map is used to access, in a constant and small time period, each node of the tree. They also propose to store, in each node, a large amount of data and particularly: a unique identifier (id), its parent id and its children id list. Thanks to an adapted data structure, a lot of memory may be saved. Firstly, we do not keep the node id in the node. This id is implicitly given by the offset from the root node in the data

structure. Next, as we do not need to go backward, we do not record the parent id. Moreover the list of children proposed by Huang implies the use of another linked list (one for each node) which increases the complexity and is memory consuming. These lists make us lose time during the tree creation. We replace these lists by two (integer) fields in the node: *son* the first child id and *brother* the first brother id. Then finding every children of a node i is easy and fast (all brothers may not have the same level):

```
01 child=node[i].son;              //first child
02 while(child !=0 ) child=node[child].brother; //next child
```

As the id gives us the index of a node in the data structure, we do not need the hash table any more and we save memory again. The underlying structure is also a bit different. Huang says that an array should not be used and uses a linked list instead. But a linked list has some drawbacks: it consumes time to allocate each node and does not offer a simple way to access a node (that is why they are obliged to use a hash table). We use instead an intermediate paginated structure: we do not link nodes but a group of nodes (a page). Each group of nodes is recorded in an array (of N elements). This simplifies the allocation because nodes are allocated by blocks. Elements can also be simply accessed: to get the i^{th} element we compute the page number $page = i \,/\, N$ and the offset in the page $offset = i \; modulo \; N$.

Now that we have seen data structures, let us see the algorithm itself which is strongly inspired from [9]. To build the tree, a priority *lifo* structure is needed with standard functions: $push(e, p)$ which pushes an element e with the specified priority p, $pop()$ which provides the last element with the highest priority level and $get\_higher\_level()$ which gives the highest priority available.

The creation of the maxtree is based on a flooding process and starts on a pixel that has the minimum value. During the flooding process, from pixel p, when a new pixel p_next is flooded, three different situations may occur (figure 3):

- A) p_next has a level higher than p (line 14). Then, a new branch of the tree is created starting from branch p (function $next\_label\_higher()$),
- B) p_next has a level lower than p and this level is unknown (line 17). Then a new node is created at the correct level in the same branch of node p and adds this pixel to this node (function $next\_label\_lower()$).
- C) p_next has a level lower than p and this level is known (line 19). Then p_next is added to the node ancestor to node p at the correct level,

The tree is built by the following algorithm called with the min value of the pixel in the lifo. To simplify the readability some variables are used as if they were globally declared. Particularly, the tree, which is the main output of the function $flood$, is split into two indexed variables: *son* and *brother* ($son[i]$ gives the first child of node i and $brother[son[i]]$ gives his second and so on...). $attributes[i]$ and $level[i]$ give respectively the value of the attribute and the gray level of node i. Notice that the index of a node is the label used to fill in a region in lab_img, allowing to link the tree node with the corresponding region. Finally, the variable $next\_label$ always gives the next index available.

```
01 void flood(img, lab_img, labs_branch, level)
02 in
03  img: the input image,
04  labs_branch: labs_branch[l] gives the label of level l in the current branch of the tree
05  level: the level of the starting point of the flooding process
06 out
07  the labeled image (lab_img) and the tree (son, brother, attributes and level)
08 {
09  index: the label;
10  p: the current pixel;
11  while( (get_higher_level()>=level) && (p=pop())!=-1)
12  {
13    for all pixels p_next, unlabeled neighbor of p {
14      if (img[p_next]>img[p]) {                       // CASE A
15        for(j=img[p]+1;j<img[p_next];j++) labs_branch[j]=0;
16        index=lab_img[p_next]=labs_branch[j]=
                 next_label_higher(img[p], img[p_next], labs_branch);
17      } else if (labs_branch[img[p_next]]==0) {              // CASE B
18        index=lab_img[p_next]=
                 labs_branch[img[p_next]]=next_label_lower(img[p_next], labs_branch);
19      } else index=lab_img[p_next]=labs_branch[img[p_next]];      // CASE C
20      // *** update here attributes[index] according to p_next ***
21      push(p_next, img[p_next]);
22      if (img[p_next]>img[p]) flood(img, lab_img, labs_branch, img[p]);
23    }
24  }
25 }
26
27 label next_label_lower(value, labs_branch)
28 in:
29  value: the level of the pixel,
30  labs_branch: labs_branch[l] gives the label of level l in the current branch of the tree
31 {
32  level[next_label]=value;
33  l=level_of_parent_of(labs_branch[value]);
34  son[next_label]=son[labs_branch[l]];
35  son[labs_branch[l]]=next_label;
36  brother[next_label]=brother[son[next_label]];
37  brother[son[next_label]]=NO_BROTHER;
38  // *** init here attributes[next_label] ***
39  return next_label++;
40 }
41
42 label next_label_higher(parent_value, value, labs_branch)
43 in:
44  parent_value: the level of the ancestor pixel,
45  value: the level of the pixel,
46  labs_branch: labs_branch[l] gives the label of level l in the current branch of the tree
47 {
48  level[next_label]=value;
49  brother[next_label]=son[labs_branch[parent_value]];
50  son[labs_branch[parent_value]]=next_label;
51  // *** init here attributes[next_label] ***
52  return next_label++;
53 }
```

A comparison with a non optimized version of the algorithm proposed in [9] (provided by the authors) shows that our implementation is in average 33% faster. This difference seems to be mainly due to memory management.

3.2 Computing Ultimate Opening

An attribute opening γ_i discards regions with attributes j smaller than i. γ_i can easily be deduced from the previously computed maxtree by pruning branches

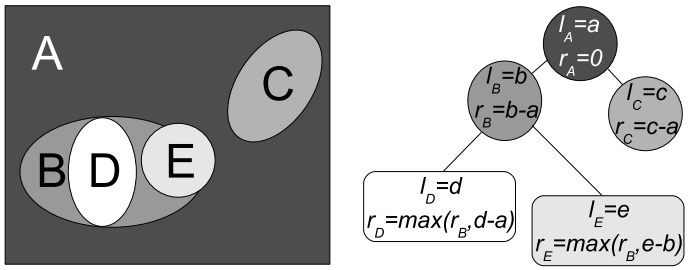

Fig. 4. The computation of the residue in the tree according to an attribute height opening

with an attribute smaller than i [7] (each node is valuated with the attribute value of its corresponding region). A residue can be computed between two successive openings γ_i and γ_{i+1} by the difference of the resulting images of each opening. In the tree, this residue is computed (in every node removed by γ_{i+1}) by the difference between the gray level of each node and its first ancestor with a different attribute. A node with attribute different from i has no residue at step i (i.e $r_i = 0$). The *U.O.* analyzes the residue of successive growing size openings and the highest residue is kept (Fig. 4, Eq. 1-2). During the process, an opening γ_i may generate a residue r_1 for a node n_1. Later, a bigger opening γ_j will generate a residue r_2 for a node n_2 ancestor of n_1 in the tree. As n_2 encompasses n_1, if r_2 is bigger than r_1, r_2 must also be assigned to n_1 (if not n_1 keeps r_1). To find the highest residue of all nodes, each ancestor n_k of each node must be checked. This is why the maxtree data structure is suitable: the highest residue will not be searched among ancestors but all ancestors will transmit their own maximum residue. The tree is recursively processed: on a given node, the computed residue is transmitted to every child (see *l 22-25* of pseudo-code). Every child will compare its own residue (*var. contrast*) with the one transmitted by its ancestors (*l 11-17*), and will keep the maximum of them (*transform_node_LUT*) as well as the opening attribute size associated with the maximum residue (*associate_node_LUT*) (*l 18-19*). This maximum is transmitted again and so on. This process is in o(n), with n the number of nodes.

Figure 4 illustrates an example of ultimate opening computation on the max-tree. The height attribute is chosen for the example (we note H_k the height of region k). First, the max-tree is created and each node is given the gray level and the attribute of the corresponding region. Note that regions B and D have the same attribute. Both regions are then removed by the same opening, of size $H_B + 1$ ($= H_D + 1$). The residue generated on region D by γ_{H_B+1} is not $l_D - l_B$ (the gray level difference with its parent node) but $l_D - l_A$, A been its first ancestor with a different attribute.

Let us process the tree to compute the ultimate height opening. The process follows a depth-first traversal starting from the root node. The first computed residue is residue r_B of region B. This residue is equal to the difference of the level of region B and the level of region A ($l_B - l_A$). Then, this residue r_B

is transmitted to child D. The residue of this child is then computed as the difference between region D level and region A level ($l_D - l_A$ and not $l_D - l_B$ as region B and region D have the same attribute and are then removed by the same opening). r_D value is the highest value between r_B and $l_D - l_A$. The next child E is processed. The residue of this child is $l_E - l_B$ and r_E is the max between r_B and $l_E - l_B$. The last region C is processed. Residue r_C is simply the difference between region C level and region A level ($l_C - l_A$). r value for all nodes is then known. Every time a residue is selected, the size of the opening leading to the residue is recorded (and transmitted to children) in order to generate the associated function q of equation 1.

```
01 void compute_uo(node, max_tr, max_in, parent_attribute, parent_value, previous_value)
02 In:
03  node: the processed node
04  max_tr: the maximum of contrast of previous nodes
05  max_in: the attribute that generate max_tr
06  parent_attribute: the attribute of the parent node
07  parent_value: the value of the parent node
08  previous_value: the value of the 1st ancestor with a
                                        different attribute: the parent of the branch
09 {
10  contrast=(attributes[node]==parent_attribute)?
                                level[node]-previous_value:level[node]-parent_value;
11  if (contrast>=max_tr) {
12    max_contrast=contrast;
13    linked_attributes=attributes[node];
14  } else {
15    max_contrast=max_tr;
16    linked_attributes=max_in;
17  }
18  transform_node_LUT[node]=max_contrast;
19  associated_node_LUT[node]=linked_attributes+1;
20  child=son[node];
21  if (attributes[node]==parent_attribute) pv=previous_value; else pv=parent_value;
22  while (child!=0) {
23    compute_uo(child, max_contrast, linked_attributes, attributes[node], level[node], pv);
24    child=brother[child];
25  }
26 }
```

3.3 Generating the Output Images

The last step consists in generating the two results (the transformation and the associated function). This is an easy step: by the use of the labeled image obtained by the flooding step and the two look-up tables computed in *compute_uo* ($transform\_node\_LUT$ and $associated\_node\_LUT$ which give, for a given label, respectively the maximum contrast and the associated attribute for this contrast), we can deduce the transformation and associated function images:

```
01  for all pixel p {
02    transform[p]=transform_node_LUT[lab_img[p]];
03    associate[p]=associated_node_LUT[lab_img[p]];
04  }
```

3.4 Iterated Ultimate Opening

The maxtree is well adapted to process *U.Os.*. Several filters may be deduced from the same tree structure. These filters can be applied on the entire image

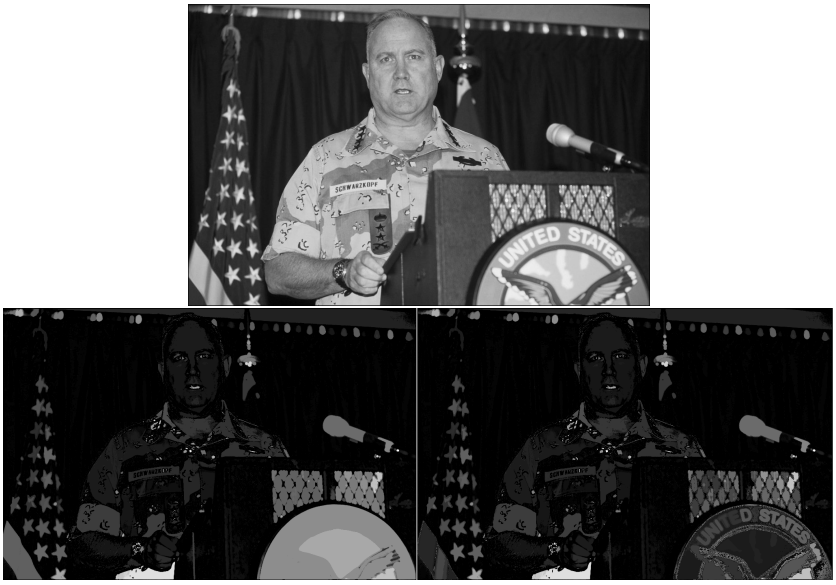

Fig. 5. From left to right: input image, the result of the *U.O.* (some details such as the text "UNITED STATES" are masked), and the result of the iterated *U.O.* where the text is clearly visible (contrast of both images is enhanced to be visible). With the maxtree, negligible additional time is needed to iterate the *U.O.* onto some areas of the image. Here, the condition to re-start *U.O.* is based on feature size and contrast.

or on a part of it (which corresponds to a branch of the tree). In this section we illustrate this property. A major issue with the *U.O.* is that sometimes, when an interesting area of the image is surrounded by a highly contrasted border (text over a sign board for example), the content of this area is masked by the *U.O.* (Fig. 5). To solve this issue, we offer to iterate the *U.O.* on such an area (Fig. 5). Function *compute_uo* can simply be adapted to this improvement and is modified in *compute_uo_iterative* to process specific areas of the image (ie. a specific branch) from the rest of the image. A condition is checked before processing branches:

```
1 if (specific condition) max_tr_propag=0; else max_tr_propag=max_contrast;
2 while (child!=0) {
3    compute_uo_iterative(child, max_tr_propag, linked_attributes,
                                          attributes[node], level[node], pv);
4    child=brother[child];
5  }
```

Instead of transmitting the estimated contrast deeper in the branch, we just transmit 0 as the previous contrast. The branch will be processed as a new tree without consuming additional time. Trying to perform this operation directly on the image is much harder and much more time consuming. The main difficulty with this approach is to determine when the *U.O.* will be restarted but this question is out of the scope of this article and depends on the application.

Format	128x128	256x256	512x512	1024x1024	2048x2048
Nb of pixels	16384	65536	262144	1048576	4194304
Time (ms)	0,18	2,39	12,01	52,04	235,53

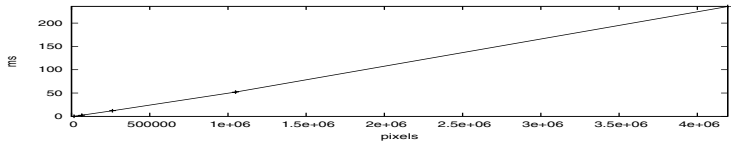

Fig. 6. Execution time of the *U.O.* according to the number of pixels

4 Results

To measure the efficiency of the implementation, we have tested it on a personal image database (about 570 various photos). We evaluate the average time consumed by our implementation by computing ten times the *U.O.* on every picture. We perform the test at different scales. The result is approximately linear according to the number of pixels in the image (Fig. 6) (except for very small images; the cache may introduce a bias). All tests have been carried out on a DELL D630 laptop with 2,4GHz T7700 processor and the implementation is in C. Given times include all the process: all allocations (lifo structure...) maxtree creation, *U.O.* process, result generation and memory cleaning (intermediary data such as the maxtree, the lifo structure...). Only I/O operations are not included in given times. Moreover, let us consider the execution time distribution: 72% of time in average is spent to build the tree, 9% and 19% of time is spent for processing the tree and generating the result respectively. This new implementation of the *U.O.* operator is a major improvement and makes this operator usable in various contexts and even in real time for rather large images.

5 Conclusion

We have presented a fast implementation of the *U.O.* based on a maxtree. Even if existing maxtree implementations may be used for our purpose, we propose a new implementation with a more efficient memory management.

As stated in the literature, several connected operators may be deduced from the same tree-structure. This is why, the maxtree is an efficient representation to implement an ultimate attribute opening operator (which definition involves a series of connected openings). The maxtree creation itself, remains the most time consuming task of our process. Moreover, we propose an iterated version of the ultimate opening. It provides more details in a given region, with negligible additional time, exploiting the re-usability of the maxtree representation.

At the end, we evaluate the executing time and show that the algorithm is quasi-linear according to the number of pixels. The *U.O.* is a powerful nonparametric tool. This fast implementation makes it very competitive and usable in real time.

Acknowledgments. We thank Pr. Salembier for providing us his maxtree implementation to perform comparisons. This work is supported by the French ANR (Agence Nationale de la Recherche) in the itowns project.

References

1. Retornaz, T., Marcotegui, B.: Scene text localization based on the ultimate opening. In: International Symposium on Mathematical Morphology, vol. 1, pp. 177–188 (2007)
2. Beucher, S.: Numerical residues. Image Vision Computing 25(4), 405–415 (2007)
3. Breen, E.J., Jones, R.: Attribute openings, thinnings, and granulometries. Computer Vision and Image Understanding 64(3), 377–389 (1996)
4. Retornaz, T.: Détection de textes enfouis dans des bases d'images généralistes. Un descripteur sémantique pour l'indexation. PhD thesis, Ecole Nationale Suprieure des Mines de Paris - C.M.M., Fontainebleau - France (October 2007)
5. Retornaz, T., Marcotegui, B.: Ultimate opening implementation based on a flooding process. In: The 12th International Congress for Stereology (September 2007)
6. Garrido, L.: Hierarchical Region Based Processing of Images and Video Sequences: Application to Filtering, Segmentation and Information Retrieval. PhD thesis, Universitat Politnica de Catalunya - Department of Signal theory and Communications, Barcelona, Spain (April 2002)
7. Salembier, P., Garrido, L.: Connected operators based on region-tree pruning strategies. In: International Conference on Pattern Recognition, ICPR 2000, September 2000, vol. 3, pp. 3371–3374 (2000)
8. Meijster, A., Wilkinson, M.H.: A comparison of algorithms for connected set openings and closings. IEEE Trans. on Pattern Analysis and Machine Intelligence 24(4) (April 2002)
9. Salembier, P., Oliveras, A., Garrido, L.: Anti-extensive connected operators for image and sequence processing. IEEE Trans. on Image Processing 7(4), 555–570 (1998)
10. Huang, X., Fisher, M., Smith, D.: An efficient implementation of max tree with linked list and hash table. In: Proc. VIIth Digital Image Computing: Techniques and Applications, pp. 299–308 (2003)
11. Najman, L., Couprie, M.: Building the component tree in quasi-linear time. IEEE Transactions on Image Processing 15(11), 3531–3539 (2006)