Milena: Write Generic Morphological Algorithms Once, Run on Many Kinds of Images

Roland Levillain[1,2], Thierry Géraud[1,2], and Laurent Najman[2]

[1] EPITA Research and Development Laboratory (LRDE)
14-16, rue Voltaire, FR-94276 Le Kremlin-Bicêtre Cedex, France
[2] Université Paris-Est, Laboratoire d'Informatique Gaspard-Monge, Équipe A3SI,
ESIEE Paris, Cité Descartes, BP 99, FR-93162 Noisy-le-Grand Cedex, France
{roland.levillain,thierry.geraud}@lrde.epita.fr, l.najman@esiee.fr

Abstract. We present a programming framework for discrete mathematical morphology centered on the concept of genericity. We show that formal definitions of morphological algorithms can be translated into actual code, usable on virtually any kind of compatible images, provided a general definition of the concept of image is given. This work is implemented in Milena, a generic, efficient, and user-friendly image processing library.

1 Introduction

Software for mathematical morphology targets several audiences: *end users*, *designers* and *providers*. *End users* of morphological tools want to apply and assemble algorithms to solve image processing, pattern recognition or computer vision problems. *Designers* of morphological operators build new algorithms by using constructs from their software framework (language, libraries, toolboxes, programs, etc.). Finally, *providers* of data structures are interested in extending their framework with new data types (images, values, structuring elements, etc.).

The size of the population of these categories is decreasing: there are more end users of morphological software than designers of algorithms, and the latter themselves outnumber providers of data structures. Morphological frameworks usually address the needs of their clients in this order, and even sometimes ignore the third or second categories. However, a full morphological framework should suit all groups of users so that structures of providers and algorithms of designers can be used by every actor. In this article, we present a software framework for mathematical morphology designed with two major goals in mind:

1. Be as simple as calling C routines for end users.
2. Be modular enough to be extended w.r.t. algorithms and data structures;

and four minor:

3. Be generic: if a morphological operator admits a general definition whatever the context (topology of the image, structuring element, etc.), then this algorithm should have a corresponding single implementation.

M.H.F. Wilkinson and J.B.T.M. Roerdink (Eds.): ISMM 2009, LNCS 5720, pp. 295–306, 2009.

4. Be close to theory: reading (and writing) algorithms should eventually become natural to scientists used to mathematical morphology notations.
5. Retain efficiency (with respect to run time speed and memory usage) when it is possible. Dedicated and efficient implementations of morphological algorithms for certain cases are known and should be selected whenever possible.
6. Be user-friendly: users should not have to address memory-related issues or deal with a program silently failing because of an arithmetic overflow. The tool should handle these situations, and help the user diagnose any problem.

The paradigm of Generic Programming (GP) [1] which is at the heart of many modern C++ libraries [2,3] and its application to the C++ language address many of these concerns. Developing a software library in the context of GP requires some effort. One of the key ideas is that such a library should be based on *abstractions* of the domain (mathematical morphology in this case). The above requirements will not be fully satisfied if we fail to reify intrinsic concepts of the domain as abstractions. Several image processing libraries relying on the GP paradigm exist (ITK [4], VIGRA [5], Morph-M [6]) but as far as the authors know, none of them seem to meet all of the above requirements.

From the general lattice theory on which is built mathematical morphology, many authors have proposed derived theoretical frameworks. The first ones are graphs [7,8], later extended to store information both *on* vertices and *between* vertices (on edges) [9,10]. The notion of *complex* (see Section 3.2) has also been used to express topological and geometrical attributes of images beyond the scope of graphs [11,12]. Generic programming frameworks to implement algorithms on complexes and grid data structures have been proposed [13,14,15]. Other possible frameworks include combinatorial maps [16] and orders [17].

Let us for instance consider the framework of graphs as the basis of morphological image processing in order to express definitions and properties as general as possible (and meet requirement 3). We could then use a graph-related library like the Boost Graph Library (BGL) [3]. However, such a design suffers from limitations, as mathematical morphology, despite having many intersections with graph theory, has its own definitions, idioms, notations, and issues. Therefore, adapting morphological algorithms to a graph software framework would distort their definitions, which is contrary to requirements 4 and 6. Moreover, we would probably lose efficiency (requirement 5) for restricted use cases in image processing (but at the same time, the most common ones): regular 2D or 3D images on grids, classical structuring elements, etc. Finally, setting graphs as the ultimate representation of images in mathematical morphology once and for all might prevent future extensions. For example the notion of complex mentioned previously, which extends the notion of graph, can be considered to form the basis of a morphological framework.

Therefore, instead of using a fixed system, we propose to rethink mathematical morphology under the light of generic programming [18]. The first step is to define software abstractions matching morphological entities (topology, sets, functions, lattices, structuring elements, geometry, etc.), starting with the concept of *discrete image*. Then, it will be possible to express algorithms in terms of

these concepts on the one hand, and provide actual data structure implementing these abstractions on the other hand.

In this paper, we present a generic and efficient C++ programming library, Milena, a part of the Olena image processing platform [19,20]. Milena uses and extends the idea of GP [21]. It implements the abstractions for mathematical morphology software mentioned previously.

This article mainly targets end users of the library and designers of algorithms. It is structured as follows: in Section 2, we study how morphological algorithms are commonly implemented and what are the issues of classical yet restrictive designs. Section 3 proposes a generic definition of an image and shows how this genericity is expressed through the image's traits. As an illustration, a small generic image processing chain is given in Section 4 and applied to various images.

2 Software Implementation of Mathematical Morphology

Translating mathematical morphology methods and objects into readable and usable algorithms is often biased either to satisfy constraints of actual data or meet software and hardware requirements. An example of the first circumstance is the prominent case of a 2-dimensional single-valued image, set on a rectangular (boxed) domain with integer coordinates (a discrete grid $D \subseteq \mathbb{Z}^2$). Many morphological algorithms are solely expressed with this framework in mind. The second bias is computer-dependent: for the sake of efficiency or simplicity of implementation, algorithms sometimes include language- or hardware-related constructs: buffers, loops, dimension decomposition, out-of-bounds behavior, etc.

Let us consider a simple example: the elementary morphological dilation of a gray-level image `ima` with a (flat) structuring element. A shortened definition in the framework of complete lattices [22] would be:

$$\delta_B(I)(x) = \sup_{h \in B} I(x + h)$$

where I (the image to process) is a function $D \to V$ associating a point from the domain D to a value from the set V; and B the structuring element associated to, e.g., the usual 4-connectivity neighborhood. A simple implementation in C++ could be as the one from Algorithm 1. However, this solution makes extra hypotheses that were not contained in the definition of the operation, e.g.:

1. The image is 2-dimensional, since it is accessed using a (row, col) notation.
2. Sites are points with nonnegative integers coordinates starting at 0.
3. The values of the image are compatible with the 8-bit `unsigned char` type.
4. The values of the image form a totally ordered set; hence the operator $<$ can be used to compute the supremum.
5. The structuring element is based on the 4-connectivity.

Each of the previous hypotheses is an actual limitation on the generality of Algorithm 1. It cannot be reused as-is if for instance one or several of the following conditions are expected:

```
image dilation(const image& input) {
  image output (input.nrows(), input.ncols()); // Initialize an output image.
  for (unsigned int r = 0; r < input.nrows() − 1; ++r) // Iterate on rows.
    for (unsigned int c = 0; c < input.ncols() − 1; ++c) { // Iterate on columns.
      unsigned char sup = input(r, c);
      if (r > 0                      && input(r−1, c) > sup) sup = input(r−1, c);
      if (r < input.nrows() − 1 && input(r+1, c) > sup) sup = input(r+1, c);
      if (c > 0                      && input(r, c−1) > sup) sup = input(r, c−1);
      if (c < input.ncols() − 1  && input(r, c+1) > sup) sup = input(r, c+1);
      output(r, c) = sup;
    }
  return output;
}
```

Algorithm 1. Non generic implementation of a morphological dilation of an 8-bit gray-level image on a regular 2D grid using a 4-c flat structuring element.

1. The input is a 3-dimensional image.
2. Its points are located on a box subset of a floating-point grid, that does not necessarily include the origin.
3. The values are encoded as 12-bit integers or as floating-point numbers.
4. The image is multivalued (e.g., a 3-channel color image).
5. The structuring element represents an 8-connectivity.

Even if the class of images accepted by Algorithm 1 covers day-to-day needs of numerous image processing practitioners, image with features from the previous list are also quite common in fields like biomedical imaging, astronomy, document image analysis or arts. Algorithm 1 also highlights less common restrictions. As is, it is unable to process images with the following features:

- A domain
  - which is not an hyperrectangle (or "box");
  - which is not a set of *points* located in a geometrical space, e.g., given a 3D triangle mesh, one can build an image by mapping each triangle to a set of values;
  - which is a restriction (subset) of another image's domain, still preserving essential properties, like the adjacency of the sites.
- A neighborhood where neighbors of a site are not expressed with a fixed-set structuring element, but through a function associating a set of sites to any site of the image. This is the case when the domain of the image is a graph, where values are attached to vertices [8].
- Non scalar image values, like color values.

Furthermore, the style used in Algorithm 1 does not allow for optimizations. An optimized code (taking advantage, for example, of a totally ordered domain of values, with an attainable upper bound), requires a whole new algorithm per compatible data structure.

```
template <typename I, typename W>
mln_concrete(I) dilation (const I& input, const W& win) {
  mln_concrete(I) output; initialize (output, input); // Initialize output.
  mln_piter(I) p(input.domain()); // Iterator on sites of the domain of 'input'.
  mln_qiter(W) q(win, p); // Iterator on the neighbors of 'p' w.r.t. 'win'.
  for_all(p) {
    accu::supremum sup = input(p); // Accumulator computing the supremum.
    for_all(q) if (input.has(q))
      sup.take(input(q));
    output(p) = sup.to_result();
    return output;
  }
}
```

Algorithm 2. Generic implementation of a morphological dilation.

In the remainder of this paper, we show how the programming framework of Milena allows programmers to easily write generic and reusable [23] image processing chains using mathematical morphology tools. For instance a Milena equivalent of Algorithm 1 could be Algorithm 2. In this algorithm `I` is a generic image type, while `W` is the type of a generic structuring element (also named *window*). `p` and `q` are objects traversing respectively the domain of `ima` and the sites of the structuring element `win` centered on `p`. The predicate `input.has(q)` ensures that `q` is a valid site of `input` (this property may not be verified e.g. when `p` is on the border of the image). `sup` iteratively computes the supremum of the values under `win` for each site `p`. An example of use similar to Algorithm 1 would be:

```
image2d<unsigned char> ima_dil = dilation(ima, win_c4p());
```

where `win_c4p()` represents the set of neighboring sites in the sense of the 4-connectivity plus the center of the structuring element.

Algorithm 2 is a small yet readable routine and is no longer specific to the aforementioned 2-dimensional 8-bit gray-level image case of Algorithm 1. It is generic with respect to its inputs, and no longer restricted by the limitations we mentioned previously. For instance it can be applied to an image defined on a Region Adjacency Graph (RAG) where each site is a region of an image, associated to an n-dimensional vector expressing features from each underlying region, provided a supremum is well defined on such a value type.

3 Genericity in Mathematical Morphology

3.1 A Generic Definition of the Concept of Image

The previous considerations about the polymorphic nature of a discrete image require a clear definition of the concept of image. To embrace the whole set of aforementioned aspects, we propose the following general definition.

Definition. An image I is a function from a domain D to a set of values V. The elements of D are called the *sites* of I, while the elements of V are its *values*.

For the sake of generality, we use the term *site* instead of *point*: if the domain of I were a RAG, it would be awkward to refer to its elements (the regions) as "points". This definition forms the central paradigm of Milena's construction. However, an actual implementation of an image object cannot rely only on this definition. It is too general as is, and mathematical morphology algorithms expect some more information from their inputs, like whether V is a complete lattice, how the neighboring relation between sites is defined, etc. Therefore, we define additional notions to supplement the definition of an image. These notions are designed to address orthogonal concerns in image processing and mathematical morphology, so that actual definitions (*implementations*) of images can be changed along one axis (e.g., the topology of D) while preserving another (e.g., the existence of a supremum for each subset $X \subseteq V$).

Algorithms are then no longer defined in terms of specific image characteristics (e.g., a domain defined as two ranges of integers representing the coordinates of each of its points) but using *abstractions* (e.g., a *site iterator* object, providing successive accesses to each site of the image, that can be deduced from the image itself). This paradigm based on Generic Programming promotes "Write Once, Reuse Everywhere Applicable" design of algorithms by introducing abstract entities (akin to mathematical objects) in software defined by their *properties*.

The *genericity* of our approach resides in both the organization of the library around entities dedicated to morphological image processing (images, sites, site sets, neighborhoods, value sets, etc.) and in the possibility to extend Milena with new structures and algorithms, while preserving and reusing existing material.

The next section presents the main entities upon which we define morphological algorithms in Milena, and how they provide genericity in mathematical morphology.

3.2 Genericity Traits

We define actual images as models of the previous definition of an image, with extra *properties* on I, D or V. These traits express the generic nature of this definition, and are related to the notions of this section. Each of them is as much orthogonal (or loosely coupled) to the others as possible, so that an actual implementation of one of these concepts can be defined and used with many algorithms regardless of the other features of the input(s). In the rest of this section, we illustrate how the limitations of Algorithm 1 mentioned in Section 2 are lifted by the generic implementation of Algorithm 2.

Restriction of the Domain. It is possible to express the restriction of an image `ima` to a subset `s` of its domain using the dedicated operator `|`; the result can then be used as input of an algorithm:

```
image2d<int> ima_dil = morpho::dilation(ima | s, win);
```

The subset `s` can either be a comprehensive collection of sites (array, set, etc.) or a predicate. A classical example is the use of a "mask" to restrict the domain of an image. This mask can for instance be a watershed line previously computed on `ima`; the dilation above would act as a reconstruction of the pixels of `ima` belonging to this watershed line.

Structuring Elements, Neighborhoods and Windows. Structuring elements of mathematical morphology can be generalized with the notion of *windows*: functions from D to $\mathcal{P}(D)$. A special case of window is a *neighborhood*: a non-reflexive symmetric binary relation on D. In the case of images set on n-dimensional regular grids (as in the previous example of dilation of a 2D image), D is a subset of $\mathbb{Z}^n$ and is expressed as an n-dimensional bounding box. Windows' members can be expressed regardless of the considered site, using a (fixed or variable) set of vectors, called *delta-sites*, as they encode a difference between two sites. For instance a 4-connectivity window is the set of 2D vectors $\{(-1,0),(0,-1),(0,0),(0,1),(1,0)\}$.

In more general cases, windows are implemented as domain-dependent functions. For instance, the natural neighbors of a site p (called the center of the window) of a graph-based image, where D is restricted to the set of vertices, are its adjacent vertices, according to the underlying graph. Such a window is implemented by an object of type `adjacent_vertices_window_p` in Milena (see below). This window does not contain delta-sites; instead, it encodes the definition of its member sites as a function of p. Using an iterator q to iterate over this window (as in Algorithm 2) successively returns each of its members.

Topological Structure. The structure of D defines relations between its elements. Classical images types are set on the structure of a regular graph, where each vertex is a site of I. More general images can be defined on general graphs, where sites can be either the vertices of the graph, its edges or even both.

An example of dilation on regular 2D image was given in Section 2. In the case of an image associating 8-bit integer values to the elements (vertices and edges) of graph, computing an elementary dilation with respect to the adjacent vertices would be written as this:

```
graph_image<int_u8> ima_dil =
  morpho::dilation(ima | vertices , adjacent_vertices_window_p());
```

`ima | vertices` creates an image based on the subset of vertices of on-the-fly, while `adjacent_vertices_window_p()` returns a window mapping each vertex to the set of its neighbors plus the vertex itself.

We can generalize this idea by using *simplicial complexes*. An informal definition of a simplicial complex (or simplicial d-complex) is "a set of simplices" (plural of simplex), where a simplex or n-simplex is the simplest manifold that can be created using n points (with $0 \leq n \leq d$). A 0-simplex is a point, a 1-simplex a line segment, a 2-simplex a triangle, a 3-simplex a tetrahedron. Simplicial complexes extends the notion of graphs; a graph is indeed a 1-complex. They can be used to define topological spaces, and therefore serve as supports for images. Figure 1 shows an example of simplicial 3-complex.

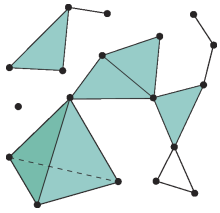


Fig. 1. A simplicial 3-complex

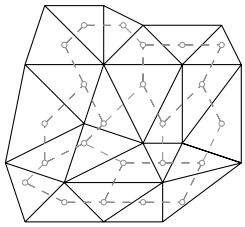


Fig. 2. A mesh seen as a simplicial 2-complex

Let us consider an image `ima` based on a simplicial 2-complex (Figure 2) where each element is located in space according to a geometry G (the notion of site location and geometry is addressed later) with 8-bit integer values. The domain D of this image is composed of points, segments and triangles. We consider a neighboring relation among triangles (also known as 2-faces) where two triangles are neighbors iff they share a common edge (1-face). The code to compute the dilation of the values associated to the triangles of D with respect to this relation is as follows:

```
complex_image<2, G, int_u8> ima_dil =
  dilation (ima | faces (2),  complex_lower_dim_connected_n_face_window_p<2, G>());
```

As in the example of the graph-based image, `ima | faces(2)` is a restriction of the domain of `ima` to the set of 2-faces (triangles). The expression `complex_lower_dim_connected_n_face_window_p<2, G>()` creates the neighboring relation given earlier (for a site p of dimension n, this window is the set of n-faces sharing an $(n-1)$-face, plus the p itself).

Site Location and Geometry. In many context, the location of the sites of an image can be independent from the structure of D. For instance if the domain of I is built on the vertices of a graph, these sites can be located in $\mathbb{Z}^n$ or $\mathbb{R}^n$ with $n \in \mathbb{N}^*$. In some cases, the location of sites is polymorphic. E.g., if D is a 3-dimensional simplicial complex located in a 3D space (as in Figure 1), the location of site p can be a 3D point (if p is a vertex), a pair or points (if p is an edge), a triplet of points (if it is a triangle) or a quadruplet (if it is a tetrahedron). We encode such information as a set of locations called a *geometry*. For instance, the term `G` from the previous code is a shortcut for `complex_geometry<2, point2d>`.

Value Set. Almost all framework support several (fixed) value sets representing mathematical entities such as $\mathbb{B}$, $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$ or subsets of them. Some of them also support Cartesian products of these sets. Not so many support user-defined value types. To be able to process any kind of values, properties should be attached to these sets: quantification, existence of an order relation, existence of a supremum or infimum, etc. Then it is possible to implement algorithms with

expected constraints on V. For instance, one can perform a dilation of a color image with 8-bit R, G, B channels by defining a supremum on the `rgb8` type:

```
rgb8 sup (const rgb8& x, const rgb8& y) {
  return rgb8(max(x.r(),y.r()), max(x.g(),y.g()), max(x.b(),y.b()));
}
image2d<rgb8> ima_dil = morpho::dilation(ima, win_c4p());
```

3.3 Design and Implementation

Milena aims at genericity (broad applicability to various inputs, reusability) and efficiency (fast execution times, minimum memory footprint). The design of the library focuses on the following features, that we can only sketch here.

Ease of Use. The interface of Milena is akin to classical C code to users, minus the idiosyncratic difficulties of the language (pointers, manual memory allocation and reclaim, weak typing, etc.). Users do not need to be C++ experts to use the library. Images and other data are allocated and released automatically and transparently with no actual performance penalty.

Efficiency. Milena handles non-trivial objects (images, graphs, etc.) through shared memory, managed automatically. The mechanism is efficient since it avoids copying data. As for algorithms, programmers can provide several versions of a routine in addition to the generic one. The selection mechanism is static (resolved at compile-time), and more powerful than function overloading: instead of dispatching with respect to *types*, it dispatches with respect to one or several *properties* attached to one or several types [21].

Usability. Milena targets both prototyping and effective image processing. In the case of very large images (1 GB), we cannot afford multiples copies of values or sometimes even loading a whole image (of e.g. several gigabytes). Therefore, the library provides alternative memory management policies to handle such inputs: in this case, memory-mapped image types which, by design, have no impact whatsoever on the way algorithms are written or called.

4 Illustrations

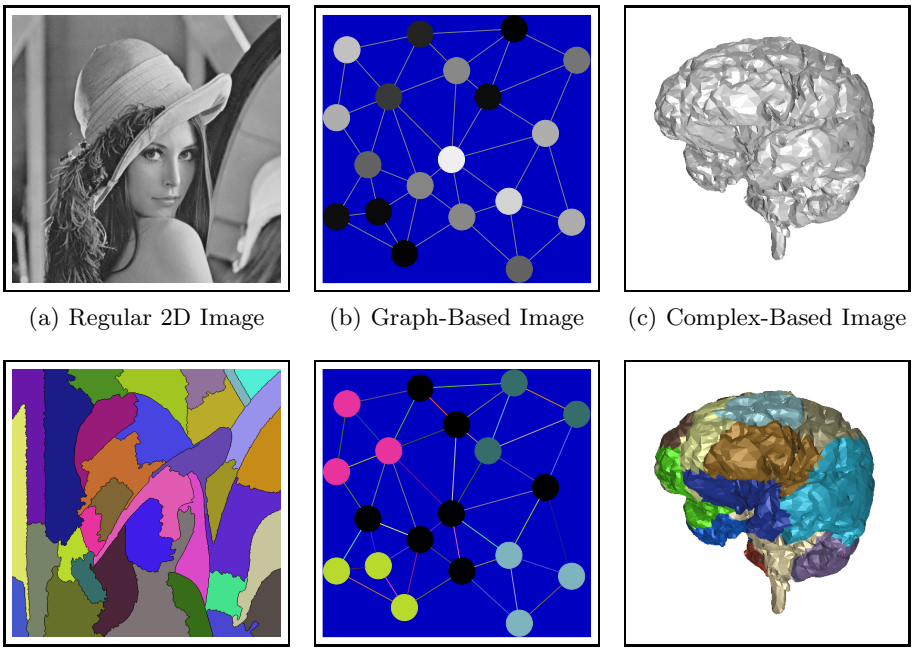
In this part, we consider a simple, classical image processing chain: from an image `ima`, compute an area closing `c` using criterion value `lambda`; then, perform a watershed transform by flooding on `c` to obtain a segmentation `s`. We apply this chain on different images `ima`. All of the following illustrations use the exact same Milena code corresponding to the processing chain above. Given an image `ima` (of type `I`), a neighborhood relation `nbh`, and a criterion value (threshold) `lambda`, this code can be written as this (`nb` is a placeholder receiving the number of catchment basins present in the watershed output image) :

```
template <typename L, typename I, typename N>
mln_ch_value(I, L) chain(const I& ima, const N& nbh, int lambda, L& nb) {
  return morpho::watershed::flooding(morpho::closing::area(ima, nbh, lambda),
                                     nbh, nb);
}
```

Regular 2-Dimensional Image. In the example of Figure 3(a), we first compute a morphological gradient used as an input for the processing chain. A 4-c window is used to compute both this gradient image and the output (Figure 3(d)), where basins have been labeled with random colors.

Graph-Based Image. Figure 3(b) shows an example of planar graph-based [7] gray-level image, from which a gradient is computed using the vertex adjacency as neighboring relation. The result shows four basins separated by a watershed line on pixels.

Simplicial Complex-Based Image. In this last example [24], a triangular mesh is viewed as a 2-simplicial complex, composed of triangles, edges and vertices (Figure 3(c)). From this image, we can compute maximum curvature values on each triangle of the complex, and compute an average curvature on edges.



(a) Regular 2D Image (b) Graph-Based Image (c) Complex-Based Image

(d) Result on a gradient of (a) (e) Result on a gradient of (b) (f) Result on the curvature of (c)

Fig. 3. Results of the image processing chain of Section 4 on various inputs

Finally, a watershed cut [25] on edges is computed, and basins are propagated to adjacent triangles and vertices for visualization purpose (Figure 3(f)).

All examples use Meyer's watershed algorithm [26], which has been proved to be equivalent to watershed cuts when used on the edges of a graph [27].

5 Conclusion

We have presented the fundamental concepts at the heart of Milena, a generic programming library for image processing and mathematical morphology, released as Free Software under the GNU General Public License. Milena allows users to write algorithms once and use them on various image types. The programming style of the library promotes simple, close-to-theory expressions.

As far as implementation is concerned, Milena extends the C++ language "from within", as a library extension dedicated to image processing. Though we designed the library to make it look familiar to image processing practitioners, it does not require a new programming language nor special tools: a standard C++ environment suffices. Moreover, as Generic Programming allows many optimizations from the compiler, the use of abstractions does not introduce actual run-time penalties.

We encourage practitioners of mathematical morphology interested in Milena to download the library at `http://olena.lrde.epita.fr/Download` and see if it can be useful to their research experiments.

Acknowledgments. The authors thank Guillaume Lazzara for his work on Milena as part of the SCRIBO project, and Alexandre Duret-Lutz for proofreading and commenting on the paper. This work has been conducted in the context of the SCRIBO project (`http://www.scribo.ws/`) of the Free Software Thematic Group, part of the "System@tic Paris-Région" Cluster (France). This project is partially funded by the French Government, its economic development agencies, and by the Paris-Région institutions.

References

1. Garcia, R., Järvi, J., Lumsdaine, A., Siek, J., Willcock, J.: A comparative study of language support for generic programming. In: Proc. of OOPSLA, pp. 115–134 (2003)
2. CGAL: Computational Geometry Algorithms Library (2008), `www.cgal.org`
3. Siek, J.G., Lee, L.Q., Lumsdaine, A.: The Boost Graph Library: User Guide and Reference Manual, 1st edn. Addison Wesley Professional, Reading (2001)
4. Yoo, T.S. (ed.): Insight into Images: Principles and Practice for Segmentation, Registration, and Image Analysis. AK Peters Ltd. (2004)
5. Köthe, U.: STL-style generic programming with images. C++ Report Magazine 12(1), 24–30 (2000)
6. Enficiaud, R.: Algorithmes multidimensionnels et multispectraux en Morphologie Mathématique: approche par méta-programmation. PhD thesis, CMM, ENSMP, Paris, France (February 2007)

7. Vincent, L.: Graphs and mathematical morphology. Signal Processing 16(4), 365–388 (1989)
8. Heijmans, H., Vincent, L.: Graph morphology in image analysis. In: Dougherty, E. (ed.) Mathematical Morphology in Image Processing, pp. 171–203. M. Dekker, New York (1992)
9. Meyer, F., Angulo, J.: Micro-viscous morphological operators. In: Proc. of ISMM, pp. 165–176 (2007)
10. Cousty, J., Najman, L., Serra, J.: Some morphological operators in graph spaces. In: Proceedings of ISMM 2009 (2009); These proceeedings
11. Bertrand, G., Couprie, M., Cousty, J., Najman, L.: Chapter title: Ligne de partage des eaux dans les espaces discrets. In: Najman, L., Talbot, H. (eds.) Morphologie mathématique: approches déterministes, pp. 123–149. Hermes Sciences (2008)
12. Loménie, N., Stamon, G.: Morphological mesh filtering and α-objects. Pattern Recognition Letters 29(10), 1571–1579 (2008)
13. Köthe, U.: Generic programming techniques that make planar cell complexes easy to use. In: Bertrand, G., Imiya, A., Klette, R. (eds.) Digital and Image Geometry. LNCS, vol. 2243, pp. 17–37. Springer, Heidelberg (2002)
14. Kettner, L.: Designing a data structure for polyhedral surfaces. In: Proc. of SCG, pp. 146–154. ACM, New York (1998)
15. Berti, G.: GrAL: the grid algorithms library. FGCS 22(1), 110–122 (2006)
16. Edmonds, J.: A combinatorial representation for polyhedral surfaces. Notices of the American Mathematical Society 7 (1960)
17. Bertrand, G., Couprie, M.: A model for digital topology. In: Bertrand, G., Couprie, M., Perroton, L. (eds.) DGCI 1999. LNCS, vol. 1568, pp. 229–241. Springer, Heidelberg (1999)
18. d'Ornellas, M.C., van den Boomgaard, R.: The state of art and future development of morphological software towards generic algorithms. International Journal of Pattern Recognition and Artificial Intelligence 17(2), 231–255 (2003)
19. LRDE: The Olena image processing library (2009), `http://olena.lrde.epita.fr`
20. Darbon, J., Géraud, T., Duret-Lutz, A.: Generic implementation of morphological image operators. In: Proc. of ISMM, Sydney, Australia, CSIRO, pp. 175–184 (2002)
21. Géraud, T., Levillain, R.: A sequel to the static C++ object-oriented programming paradigm (SCOOP 2). In: Proc. of MPOOL, Paphos, Cyprus (July 2008)
22. Goutsias, J., Heijmans, H.J.A.M.: Fundamenta morphologicae mathematicae. Fundamenta Informaticae 41(1-2), 1–31 (2000)
23. Köthe, U.: Reusable software in computer vision. In: Jähne, B., Haussecker, H., Geißler, P. (eds.) Handbook of Computer Vision and Applications. Systems and Applications, vol. 3, pp. 103–132. Academic Press, San Diego (1999)
24. Alcoverro, M., Philipp-Foliguet, S., Jordan, M., Najman, L., Cousty, J.: Region-based 3D artwork indexing and classification. In: 3DTV, pp. 393–396 (2008)
25. Cousty, J., Bertrand, G., Najman, L., Couprie, M.: Watershed cuts: minimum spanning forests and the drop of water principle. IEEE PAMI (to appear, 2009)
26. Meyer, F.: Un algorithme optimal de ligne de partage des eaux. In: Actes du 8e Congrès AFCET, Lyon-Villeurbanne, France, AFCET, pp. 847–857 (1991)
27. Cousty, J., Bertrand, G., Najman, L., Couprie, M.: On watershed cuts and thinnings. In: Coeurjolly, D., Sivignon, I., Tougne, L., Dupont, F. (eds.) DGCI 2008. LNCS, vol. 4992, pp. 434–445. Springer, Heidelberg (2008)