Writing Reusable Digital Geometry Algorithms in a Generic Image Processing Framework
Roland Levillain, Thierry Géraud, Laurent Najman

To cite this version:
Writing Reusable Digital Topology Algorithms in a Generic Image Processing Framework

Roland Levillain\textsuperscript{1,2}, Thierry Géraud\textsuperscript{1,2}, Laurent Najman\textsuperscript{2}

\textsuperscript{1} EPITA Research and Development Laboratory (LRDE)
14-16, rue Voltaire, FR-94276 Le Kremlin-Bicêtre Cedex, France
\textsuperscript{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. Digital Topology software should reflect the generality of the underlying mathematics: mapping the latter to the former requires genericity. By designing generic solutions, one can effectively reuse digital topology data structures and algorithms. We propose an image processing framework focused on the Generic Programming paradigm in which an algorithm on the paper can be turned into a single code, written once and usable with various input types. This approach enables users to design and implement new methods at a lower cost, try cross-domain experiments and help generalize results.

1 Introduction

Like Mathematical Morphology (MM), Digital Topology (DT) has many applications in image analysis and processing. Both present sound mathematical foundations to handle many types of discrete images. In fact most methods from Mathematical Morphology or Digital Topology are not tied to a specific context (image type, neighborhood, topology): they are most often described in abstract and general terms. Thus they are not limiting their field of application. However, software packages for MM and DT rarely take (enough) advantage of this generality: an algorithm is sometimes reimplemented for each image and/or each value type, or worse, written for a unique input type. Such implementations are not reusable because of their lack of genericity. These limitations often come from the implementation framework, which prohibits a generic design of algorithms. A recent and notable exception is the DGtal project, which proposes Digital Geometry (DG) software tools and algorithms built in a generic C++ framework \textsuperscript{1}.

Thanks to the Generic Programming (GP) paradigm, provided in particular by the C++ language, one can design and implement generic frameworks. This paradigm is especially well-suited to the field of scientific applications where the efficiency, widespread availability and standardization of C++ are real assets. To this end, we have designed a paradigm dedicated to generic and efficient scientific software \textsuperscript{2} and applied the idea of generic algorithms to MM in Image Processing (IP) \textsuperscript{3}, as suggested by d’Ornellas and van den Boomgaard \textsuperscript{4}. The
result of our experiments is a generic library, Milena, part of the Olena image processing platform [5].

Lamy suggests to implement digital topology in IP libraries [6]. The proposed solution, applied to the ITK library [7,8] “works for any image dimension”. In this paper, we present a framework for the generic implementation of DT methods within the Milena library, working for any image type supporting the required notions (value types, geometric and topological properties, etc.). Such a generic framework requires the definition of concepts from the domain (in particular, of an image) to organize data structures and algorithms, as explained in Sect. 2. Given these concepts it is possible to write generic algorithms, like a homotopic thinning operator making use of various definitions of the notion of simple point. We present a generic definition of such an operator in Sect. 3 and show some illustrations in Sect. 4. Section 5 concludes on the extensibility of this work along different axes: existing algorithms, new data structures and efficiency.

2 Genericity in Image Processing

In order to design a generic framework for image processing, we have previously proposed the following definition of an image [3].

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; e.g. a site could represent a triangle of a surface mesh used as the domain of an image. Classical site sets used as image domains encompass hyperrectangles (boxes) on regular $n$-dimensional grids, graphs and complexes (see Sect. 3).

In the GP paradigm, these essential notions (image, site set, site, value) must be translated into interfaces called concepts in Milena (Image, Site_Set, etc.) [9]. These interfaces contain the list of services provided by each type belonging to the concept, as well as its associated types. For instance, a type satisfying the Image concept must provide a domain() routine (to retrieve $D$), as well as a domain_t type (i.e. the type of $D$) satisfying the Site_Set concept. Concepts act as contracts between providers (types satisfying the concept) and users (algorithms expressing requirements on their inputs and outputs through concepts). For instance, the breadth_first_thinning routine from Algorithm 1.3 expects the type $I$ (of the input image) to fulfill the requirements of the Image concept. Likewise nbh must be a Neighborhood; and is_simple and constraint must be functions taking a value of arbitrary type and returning a Boolean value (Function_v2b concept).

3 Generic Implementation of Digital Topology

Let us consider the example of homotopic skeletonization by thinning. Such an operation can be obtained by the removal of simple points (or simple sites
in the Milena parlance) using Algorithm 1.1 [10]. A point of an object is said to be simple if its deletion does not change the topology of the object. This algorithm takes an object $X$ and a constraint $K$ (a set of points that must not be removed) and iteratively deletes simple points of $X \setminus K$ until stability is reached.

Algorithm 1.1 is an example of an algorithm with a general definition that could be applied to many input types in theory. But in practice, software tools often allow a limited set of such input types (sometimes just a single one), because some operations (like “is simple”) are tied to the definition of the algorithm [3].

Algorithm 1.2 shows a more general version of Algorithm 1.1 where implementation-specific elements have been replaced by mutable parts: a predicate stating whether a point $p$ is simple with respect to a set $X$ (is_simple); a routine “detaching” a (simple) point $p$ from a set $X$ (detach); and a predicate declaring whether a condition (or a set of conditions) on $p$ is satisfied before considering it for removal (constraint). The algorithm takes these three functions as arguments in addition to the input $X$. Algorithm 1.2 is a good candidate for a generic C++ implementation of the breadth-first thinning strategy and has been implemented as Algorithm 1.3 in Milena. This algorithms implements the breadth-first traversal by using a FIFO (First In, First Out) queue. The set $X$ is represented by a binary image ($V = \{\text{true}, \text{false}\}$), that must be compatible with operations performed within the algorithm. Inputs is_simple, detach and constraint have been turned into function objects (also called functors).

The breadth_first_thinning routine creates and returns an image with type mln_concrete(I); it is an image type equivalent to I that allows to store data for every sites independently (which is not the case for some image types).

Simple Point Characterization Implementation

There are local characterizations of simple points in 2D, 3D and 4D, which can lead to look-up table (LUT) based implementations [11]. However, since the number of configurations of simple and non-simple points in $\mathbb{Z}^d$ is $2^{3^{d-1}}$, this approach can only be used in practice in 2D (256 configurations, requiring a LUT of 32 bytes) and possibly in 3D (67,108,864 configurations, requiring a LUT of 8 megabytes). The 4D case exhibits $2^{80}$ configurations, which is intractable using a LUT, as it would need 128 zettabytes (128 billions of terabytes) of memory. Couprie and Bertrand have proposed a more general framework for checking for simple points using cell complexes [11] and the collapse operation. Intuitively, complexes can be seen as a generalization of graphs. An informal definition of a simplicial complex (or simplicial d-complex) is “a set of simplices” (plural of

1 In Algorithm 1.3 mln_ch_value(I, V) and mln_concrete(I) are helper macros. The former returns the image type associated to I where the value type has been set to V. The latter returns an image type corresponding to I with actual data storage capabilities. In many cases, mln_concrete(I) is simply equal to I.

2 Note that the notion of “constraint” is not the same in Algorithm 1.1 and Algorithm 1.3; in the former, it is the set of points to preserve, while in the latter is it a predicate that a candidate point must pass to be removed.
Algorithm 1.1. Breadth-First Thinning.

**Data**: $E$ (a set of points/sites), $X \subseteq E$ (initial set of points), $K \subseteq X$ (a set of points (constraint) that cannot be removed)

**Result**: $X$

$P \leftarrow \{ p \in X \mid p \text{ is simple for } X \}$

while $P \neq \emptyset$ do

$S \leftarrow \emptyset$

for each $p \in P$ do

if $p \notin K$ and $p$ is simple for $X$ then

$X \leftarrow X - \{p\}$

for each $n \in N(p) \cap X$ do

$S \leftarrow S \cup \{n\}$

$P \leftarrow \emptyset$

for each $p \in S$ do

if $p$ is simple for $X$ then $P \leftarrow P \cup \{p\}$

Algorithm 1.2. A generic version of Algorithm 1.1

**Data**: $E, X \subseteq E, N$ (neighborhood),

$\text{is\_simple}$ (a function saying whether a point is simple),

$\text{detach}$ (a routine detaching a point from $X$),

$\text{constraint}$ (a function representing a constraint)

**Result**: $X$

$P \leftarrow \{ p \in X \mid \text{is\_simple}(p, X) \}$

while $P \neq \emptyset$ do

$S \leftarrow \emptyset$

for each $p \in P$ do

if $\text{constraint}(p)$ and $\text{is\_simple}(p, X)$ then

$X \leftarrow \text{detach}(X, p)$

for each $n \in N(p) \cap X$ do

$S \leftarrow S \cup \{n\}$

$P \leftarrow \emptyset$

for each $p \in S$ do

if $\text{is\_simple}(p, X)$ then $P \leftarrow P \cup \{p\}$
Algorithm 1.3. A generic C++ implementation of Algorithm 1.2 in Milena. Functors are highlighted.

```cpp
template <typename I, typename N, typename F, typename G, typename H>
mln_concrete(I) breadth_first_thinning(const Image<I>& input_,
   const Neighborhood<N>& nbh_,
   Function_v2b<F>& is_simple_,
   G& detach,
   const Function_v2b<H>& constraint_)
{
   // Convert arguments to their exact types and initialize ‘output’.
   const I& input = exact(input_);
   const N& nbh = exact(nbh_);
   F& is_simple = exact(is_simple_);
   const H& constraint = exact(constraint_);
   mln_concrete(I) output = duplicate(input);
   is_simple.set_image(output); // Bind ‘output’ to ‘is_simple’.
   detach.set_image(output); // Bind ‘output’ to ‘detach’.

   // Step #1. Initialize a FIFO queue with simple points candidates.
   typedef mln_psite(I) psite;
   p_queue_fast<psite> queue;
   // Image ‘in_queue’ records whether a point site is in the queue.
   mln_ch_value(I, bool) in_queue;
   initialize(in_queue, input);
   data::fill(in_queue, false);
   mln_piter(I) p(output.domain());
   for_all(p) // ∀ ‘p’ ∈ the domain of ‘output’...
   {
      if (output(p) && constraint(p) && is_simple(p)) {
         queue.push(p); in_queue(p) = true; // Push ‘p’ into the queue.
      }
   }

   // Step #2. Process the FIFO queue until it is empty.
   while (!queue.is_empty()) {
      psite p = queue.pop_front(); in_queue(p) = false;
      if (output(p) && constraint(p) && is_simple(p)) {
         detach(p); // ‘p’ is simple and passes the constraint; detach it.
         // Process the neighbors of ‘p’.
         mln_niter(N) n(nbh, p);
         for_all(n) // ∀ ‘n’ in the neighborhood of ‘p’...
         {
            if (output.domain().has(n) // Prevent out-of-bound accesses.
               && output(n) && constraint(n) && is_simple(n)
               && !in_queue(n)) {
               queue.push(n); in_queue(n) = true; // Push ‘n’ into the queue.
            }
         }
      }
   }
   return output;
}
```
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. A graph is indeed a 1-complex. Figure 1(a) shows an example of a simplicial complex. Likewise, a cubical complex or cubical \( d \)-complex can be thought as a set of \( n \)-faces (with \( 0 \leq n \leq d \)) in \( \mathbb{Z}^d \), like points (0-faces), edges (1-faces), squares (2-faces), cubes (3-faces) or hypercubes (4-faces). Figure 1(b) depicts a cubical complex sample.

Complexes support a topology-preserving transformation called collapse. An elementary collapse removes a free pair of faces of a complex, like the square face \( f_1 \) and its top edge \( e_1 \), or the edge \( e_2 \) and its top vertex \( v \), in Fig. 1(b). The pair \( (f_2, e_3) \) cannot be removed, since \( e_3 \) also belongs to \( f_3 \). Successive elementary collapses form a collapse sequence that can be used to remove simple points. Collapse-based implementations of simple-point deletion can always be used in 2D, 3D and 4D, though they are less efficient than their LUT-based counterparts. On the other hand, they provide some genericity as the collapse operation can have a single generic implementation on complexes regardless of their structure.

4 Illustrations

Using this generic approach, Algorithm 1.3 can be used to compute skeletons of various input images.
4.1 Skeleton of a 2D Binary Image

Our first illustration uses a classical 2D binary image built on a square grid (Fig. 2(a)). The following lines produce the result shown on Fig. 2(b).

```c
typedef image2d<bool> I;
typedef neighb2d N;
I output = breadth_first_thinning(input,
c4(),
is_simple_point2d<I, N>(c4(), c8()),
detach_point<I>(),
no_constraint());
```

I and N are introduced as aliases of the image and neighborhood types for convenience. The `breadth_first_thinning` algorithm is called with five arguments, as expected. The first two ones are the input image and the (4-connectivity) neighborhood used in the algorithm. The last three ones are the functors governing the behavior of the thinning operator. The call `is_simple_point2d<I, N>(c4(), c8())` creates a simple point predicate based on the computation of the 2D connectivity numbers [10] associated with the 4-connectivity for the foreground and the 8-connectivity for the background. To compute these numbers efficiently, `is_simple_point2d` uses a LUT containing all the possible configurations in the 8-connectivity neighborhood of a pixel. `detach_point<I>()` is a simple functor removing a pixel by giving it the value “false”. Finally, `no_constraint` is an empty functor representing a lack of constraint.

We also present a variation of the previous example where the fifth argument passed to the function is an actual constraint, preserving all end points of the initial image (see Fig. 2(c)). This result is obtained by invoking the generic
functor \texttt{is\_not\_end\_point} in the following lines. This call creates a predicate characterizing end points by counting their number of neighbors.

\begin{verbatim}
I output_with_end_points =
breadth_first_thinning(input,
c4(),
is_simple_point2d\langle I, N\rangle(c4(), c8()),
detach_point\langle I\rangle(),
is_not_end_point\langle I, N\rangle(c4(), input));
\end{verbatim}

\section*{4.2 Skeleton of a 3D Binary Image}

This second example in 3D is similar to the previous one in 2D. The domain of the image is a box on a cubical grid; the 26- and the 6-connectivity are respectively used for the foreground and the background. The output of Fig. 3(b) is obtained from the 3D volume shown in Fig. 3(a) with the following lines.

\begin{verbatim}
typedef image3d\langle bool \rangle I;
typedef neighb3d N;
I output =
breadth_first_thinning(input,
c26(),
is_simple_point3d\langle I, N\rangle(c26(), c6()),
detach_point\langle I\rangle(),
noc_constraint());
\end{verbatim}
Fig. 4. Computation of a skeleton using breadth-first thinning. The triangle mesh surface (a) (35,286 vertices and 70,568 triangles) is seen as a simplicial 2-complex. The image of curvature (b) is computed on the edges of the mesh, and simplified using an area opening filter. All curvature regional minima are then removed from the mesh, and the skeleton (c) is obtained with Algorithm 1.3 using the collapse operation.

The only real difference with the previous example is the use of the functor is_simple_point3d. The default implementation of this predicate uses an on-the-fly computation of 3D connectivity numbers. We have also implemented a version based on a precomputed LUT which showed significant speed-up improvements.

Please note that the predicates is_simple_point2d and is_simple_point3d are specifically defined for a given topology in order to preserve performances.

4.3 Thick Skeleton of a 3D Mesh Surface

In this third example, we manipulate discrete mesh surfaces composed of triangles. The input of the thinning operator is a surface containing “holes”, obtained from the mesh shown in Fig. 4(a) by removing triangles located in regional minima of the surface’s curvature (darkest areas of Fig. 4(b)). The result presented in Fig. 4(c) is obtained with the following lines. Types are not shown to make this code more readable.

```
output = breadth_first_thinning(input,
   nbh,
   is_simple_triangle,
   detach_triangle,
   no_constraint());
```

In the previous code, input is a triangle-mesh surface represented by an image built on a simplicial 2-complex and nbh represents an adjacency relationship between triangles sharing a common edge. Function objects is_simple_triangle and detach_triangle are operations compatible with input’s type; they are generic routines based on the collapse operation mentioned in Sect. 3 working with any complex-based binary image.
The input image is constructed so that the sites browsed by the for_all loops in Algorithm 1.3 are only 2-faces (triangles), while preserving access to values at 1-faces and 0-faces. Thus, even though they receive 2-faces as input parameters, is_simple_triangle and detach_triangle are able to inspect the adjacent 1-faces and 0-faces and determine whether and how a triangle can be completely detached from the surface through a collapse sequence.

The resulting skeleton is said to be thick, since it is composed of triangles connected by a common edge. The corresponding complex is said to be pure, as it does not contain isolated 1-faces or 0-faces (that are not part of a 2-face).

### 4.4 Thin Skeleton of a 3D Mesh Surface

To obtain a thin skeleton, we can use a strategy based on successive n-collapse operations, with n decreasing [12]. From the input of the previous example, we can obtain a ultimate 2-collapse by removing all simple pairs composed of a 2-face and a 1-face (a triangle and an adjacent edge). The following lines compute such an ultimate 2-collapse. The iteration on input’s domain is still limited to triangles (2-faces).

```plaintext
collapse2 = breadth_first_thinning(input, nbh,
   is_triangle_in_simple_pair,
   detach_triangle_in_simple_pair,
   no_constraint());
```

Functor is_triangle_in_simple_pair checks whether a given triangle is part of a simple pair, and if so detach_triangle_in_simple_pair is used to remove
the pair. Thinning the initial surface with this “simple site” definition produces
a mesh free of 2-faces (triangles), as shown in Fig. 5(a).

From this first skeleton, we can compute an ultimate 1-collapse, by removing
all simple pairs composed of an edge (1-face) and a vertex (0-face). This skeleton
is produced with the following code, where input2 is an image created from
collapse2, and for which the domain of has been set to the edges of the complex,
(instead of the triangles).

```cpp
collapse1 = breadth_first_thinning(input2,
    nbh,
    is_edge_in_simple_pair,
    detach_edge_in_simple_pair,
    no_constraint());
```

Here is_edge_in_simple_pair and detach_edge_in_simple_pair respectively
test and remove an edge along with a vertex that form a simple pair. The result
is a simplified skeleton, with no isolated branches, as the lack of constraint
(no_constraint) does not preserve them. The output of the ultimate 1-collapse
on the bunny mesh is depicted in Fig. 5(b). It contains the crest lines that form
the boundaries of catchment basins, such as in the watershed transform, and, in
addition, the crest lines that make the previous ones connect one to another.

Note that in both cases, the neighborhood object nbh is the same, as it
represents the adjacency of two n-faces connected by a common adjacent (n−1)-
face. In the case of the 2-collapse, the neighborhood of a site (triangle) is the set
of adjacent triangles connected by an edge, while in the case of the 1-collapse,
the neighborhood of a site (edge) is the set of adjacent edges connected by a
vertex.

4.5 Execution Times

Table 1 shows the execution times of the previous illustrations, computed on
a PC running Debian GNU/Linux 6.0.4, featuring an Intel Pentium 4 CPU
running at 3.4 GHz with 2 GB RAM at 400 MHz, using the C++ compiler g++
(GCC) version 4.4.5, invoked with optimization option ‘-03’. The first three test
cases use a simple point criterion based on connectivity numbers, while the last
three use a collapse-based definition.

5 Conclusion

We have presented building blocks to implement reusable Digital Topology al-
gorithms in an Image Processing framework, Milena. Given a set of theoretical
constraints on its inputs, an algorithm can be written once and reused with many
compatible image types. This design has previously been proposed for Math-
ematical Morphology, and can be applied to virtually any image processing field.
Milena is Free Software released under the GNU General Public License, and
can be freely downloaded from [http://olena.lrde.epita.fr/](http://olena.lrde.epita.fr/).
Table 1. Execution times of Algorithm 1.3 for various inputs. Figures correspond to the time spent in the breadth_first_thinning routine only.

<table>
<thead>
<tr>
<th>Input</th>
<th>Input size</th>
<th>Constraint</th>
<th>Output</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>2D image (Fig. 2(a))</td>
<td>321 × 254 pixels</td>
<td>None</td>
<td>Fig. 2(b)</td>
<td>0.08 s</td>
</tr>
<tr>
<td>2D image (Fig. 2(a))</td>
<td>End points</td>
<td>Fig. 2(c)</td>
<td>0.10 s</td>
<td></td>
</tr>
<tr>
<td>3D image (Fig. 3(a))</td>
<td>41 × 41 × 41 voxels</td>
<td>None</td>
<td>Fig. 3(b)</td>
<td>2.67 s</td>
</tr>
<tr>
<td>Mesh (2-faces only) (Fig. 4(a))</td>
<td>35,286 0-faces + 105,852 1-faces +</td>
<td>None</td>
<td>Fig. 4(c)</td>
<td>159.53 s</td>
</tr>
<tr>
<td>Mesh (2- and 1-faces) (Fig. 4(a))</td>
<td>35,286 0-faces + 105,852 1-faces +</td>
<td>None</td>
<td>Fig. 5(a)</td>
<td>68.78 s</td>
</tr>
<tr>
<td>Mesh (1- and 0-faces) (Fig. 5(a))</td>
<td>70,568 2-faces</td>
<td>None</td>
<td>Fig. 5(b)</td>
<td>46.18 s</td>
</tr>
</tbody>
</table>
tem@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


