Generic Programming Techniques that Make Planar Cell Complexes Easy to Use

Ullrich Köthe

University of Hamburg, Cognitive Systems Group
koethe@informatik.uni-hamburg.de

Abstract: Cell complexes are potentially very useful in many fields, including image segmentation, numerical analysis, and computer graphics. However, in practice they are not used as widely as they could be. This is partly due to the difficulties in actually implementing algorithms on top of cell complexes. We propose to use generic programming to design cell complex data structures that are easy to use, efficient, and flexible. The implementation of the new design is demonstrated for a number of common cell complex types and an example algorithm.

1 Introduction

Cell complexes and related structures (such as combinatorial maps) are fundamental concepts of digital topology. They are potentially very useful in many application areas. For example, they can be used in image analysis to represent a segmented image, in computational geometry and computer graphics to model 2- and 3-dimensional objects, and in numerical analysis to build finite element meshes. Yet, in practice they are not used as widely as one would expect. I believe that this is partly due to the difficulties their actual implementation and application poses to the programmer. In this paper I will present a solution to this problem that was originally developed in the context of image analysis. It is interesting to note that independent investigations in other fields (e.g. computational geometry and numerical analysis) arrived at very similar solutions [2,5]. Although these fields have quite different goals, the generic concepts to be described here seem to be equally helpful.

In the field of image analysis, image segmentation is the most important application for cell complexes. On the basis of segmented images, questions like "What are the connected regions?", "Where do regions meet?", "Where are the edges?" etc. will be answered. It is well known that naive representations of segmentation results, such as binary images or edge images, have problems in answering such questions consistently and without topological contradictions.

The most infamous problem is the so called *connectivity paradox* which occurs when one defines connected regions by determining connected components in a binary image. No matter whether 8- or 4-neighborhood is used while constructing the connected components, there are configurations that violate Jordan's curve theorem. That is, in this representation it is not always assured that a closed curve partitions the plane into exactly two distinct connected components (see e.g. [8]).

Kovalevsky [8] has shown that problems like this can be overcome by using topological cell complexes (formal definition below). In particular, planar cell complexes (i.e., 2-complexes that can be embedded in the plane) are appropriate for image analysis. On a theoretical level, researchers generally agree on the usefulness of cell complex representations, but in practice they are quite rarely used. Instead, most segmentation methods deal with topological problems heuristically, if at all. I'll demonstrate the difficulties in practical application of cell complexes by means of two typical examples: the cell complexes defined in the TargetJr image analysis framework [13] and the combinatorial maps defined by Dufourd and Puitg [3]. These examples show that current software either requires a lot of manual work to build and manipulate a cell complex or are very slow in execution.

The first example are the topological classes of TargetJr [13]. TargetJr is a very popular basic framework for image analysis research and, to a degree, represents the state-of-the-art in object-oriented image analysis software. It is written in C++ and thus allows for very fast algorithms and a moderate degree of flexibility. In the area of topology, it provides basic building blocks for 2-dimensional cell complexes, most notably: `Vertex`, `Edge`, `Face`, `OneChain`. The former three represent the different cell types, while the latter encodes a sequence of connected edges, which e.g. represent the boundary of a face. In principle, it is possible to construct any planar cell complex by using these primitives. But in practice this is very hard, because the user is responsible for ensuring consistency of the resulting complex – there is no class `CellComplex` that would help the programmer in constructing a topological structure. In the words of the TargetJr documentation:

> "For operations that must reuse the vertices [...] the programmer is responsible for maintaining their consistency with the topological structure as a whole." [13], http://www.targetjr.org/manuals/Topology/node18.html

Handling is further complicated by the lack of separation between the combinatorial and geometrical structures of the complex – all topology classes also carry geometric information (e.g. coordinates in case of vertices). Furthermore, when the cell complex has finally been constructed, navigation on it is also quite hard. For example, there is no obvious way of getting the ordered cycle of edges around each vertex. Essentially, TargetJr thus leaves the most difficult tasks to the user.

A totally different approach to the construction of cell complexes is described by Dufourd and Puitg [3], my second example. They use the notions of *dart* and *orbit* to define *quasi-maps*, which contain planar maps (planar cell complexes) as special cases (formal definitions below). This theory is formally specified and implemented in a functional programming languages, which leads to a very elegant and easy to use solution. For example, the orbits can be retrieved by simple functions, so that navigation on the map is very easy. Since all required constraints are automatically enforced during map construction, inconsistent structures cannot be build. This takes a lot of responsibility from the user.

However, this approach also has some disadvantages. First, it is very difficult to integrate it with other image analysis software: most image analysis software isn't written in a functional language, and it is difficult to connect software components written in different paradigms and languages. Second, according to the definitions in [3] even the simplest operations (getting the successor of a dart in an orbit) take *linear*

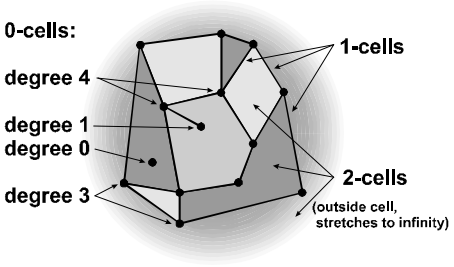

Fig. 1. Example for a planar cell complex. Some cells are marked with their types. The degree (= number of 1-cells bounded) is given for some 0-cells.

time in the size of the cell complex. This leads to unacceptably slow algorithms when cell complexes of realistic size (thousands of faces) need to be processed. It is not clear to me whether a more efficient implementation (with constant complexity in the basic operations) is possible in a functional environment.

This paper proposes to use *generic programming* in order to overcome the problems mentioned. It describes possibilities to define operations on cell complexes in a way that is simultaneously efficient, flexible and easy to use. These definitions will take on the form of *abstract interfaces*. It will also be shown how these interfaces can be implemented on top of different topological data structures.

2 Cell Complexes and Related Topological Structures

Discrete topological structures can be defined in multiple ways, giving rise to different data structures. However, from the *viewpoint of algorithms* that are to operate on top of those data structures, the differences are not very significant: many topological data structures provide essentially very similar functionality to algorithms. This is the basic justification for our attempt to define a uniform abstract interface.[1]

Let us briefly review some common definitions of digital topology (detailed treatment is beyond the scope of this paper). The best known is the *cellular complex* [8]:

> **Definition 1:** A cell complex is a triple (Z, dim, B) where Z is a set of *cells*, *dim* is a function that associates a non-negative integer *dimension* to each cell, and $B \subset Z \times Z$ is the *bounding relation* that describes which cells bound other cells. A cell may bound only cells of larger dimension, and the bounding relation must be transitive. If the largest dimension is k, we speak of a *k-complex*.

Two cells are called *incident* if one bounds the other. The bounding relation is used to define open sets and thus induces a topology on the cell complex: a set is called *open* if, whenever a cell z belongs to the set, all cells bounded by z do also belong to the set. In the context of image segmentation we are particularly interested in cell complexes with maximum dimension 2 which can be embedded in the plane. We will call these *planar cell complexes*. Figure 1 shows a planar cell complex. We will use the terms *node*, *edge*, and *face* synonymously to 0-, 1-, and 2-cells respectively.

---

[1] The correspondences between the considered models will be formally studied elsewhere.

Another popular definition is that of a *combinatorial map* [3]. This definition will prove especially useful for our interface definition and implementation:

Definition 2: A *combinatorial map* is a triple (D, σ, α) where D is a set of *darts* (also known as *half-edges*), σ is a permutation of the darts, and α is an involution of the darts.

In this context, a *permutation* is a mapping that associates to each dart a unique predecessor and a unique successor. By following successor chains, we can identify the *cycles* or *orbits* of the permutation. An *involution* is a permutation where each orbit contains exactly two elements. It can be shown that the mapping $\varphi = \sigma^{-1}\alpha$ is also a permutation. The correspondence between combinatorial maps and cell complexes is established as follows: we associate each orbit in σ with a 0-cell (node), each orbit in α with a 1-cell (edge; an edge is thus a pair of half-edges), and each orbit in φ with a 2-cell (face) of a cell complex. A thus defined cell bounds another one (of higher dimension), if their corresponding orbits have a dart in common.

In the context of rectangular rasters, Khalimsky's topology is very useful [6]:

Definition 3: The Khalimsky topology of the 1-dimensional discrete line is obtained by interpreting every other point as an open or closed set respectively. The topology of a 2-dimensional rectangular raster is constructed as the product topology of a horizontal and vertical topological line.

To interpret the topological line in the terminology of cell complexes we associate dimension 1 to the open points and dimension 0 to the closed points. The dimensions in the topological raster are obtained as the sum of the vertical and horizontal dimensions. Each cell bounds the cells in its 8-neighborhood that have lower dimension. Thus, a 0-cell bounds all its 8-neighbors, and a 1-cell bounds two 2-cells. Figure 2 (left) shows a raster defined like this.

Finally, when it comes to iterative segmentation and irregular pyramids, the *block complex* is a very useful notion. The idea is to build a new cell complex on top of an existing one by merging cells into larger *block cells*, or *blocks* for short. Blocks are defined by the property that they either contain a single 0-cell, or are homeomorphic to an open *k-sphere*. (A block is homeomorphic to an open k-sphere if it is isomorphic to a simply connected open set in some k-complex – see [9] for details.) A block's dimension equals the highest dimension of any of its cells.

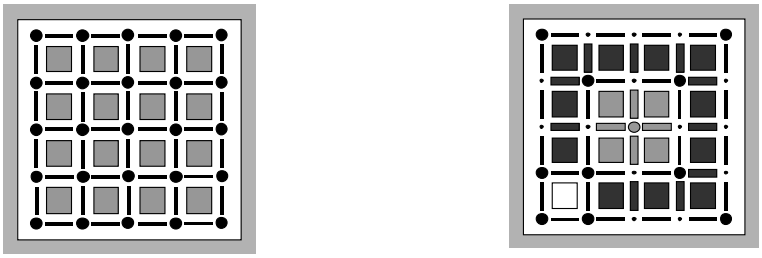

Fig. 2. left: Khalimsky topology on a square raster (0-cells: black balls, 1-cells: black lines, 2-cells: gray squares); right: block complex defined on top of a Khalimsky raster (0-blocks: large black balls, 1-blocks: black lines and small balls, 2-blocks: gray squares, rectangles and balls)

Definition 4: A *block* complex is a complete partition of a given cell complex into blocks. The bounding relation of the blocks is induced by the bounding relation of the contained cells: a block bounds another one iff it contains a cell that bounds a cell in the other block.

Note that the induced bounding relation must meet the requirements of definition 1, i.e. blocks may only bound blocks of smaller dimension, and the relation must be transitive. This poses some restrictions on valid partitions. In particular, all 1-blocks must be sequences of adjacent 0- and 1-cells, and all junctions between 1-blocks must be 0-blocks. See figure 2 right for an example.

3 Generic Programming

Generic programming is a new programming methodology developed by A. Stepanov and D. Musser [12]. It is based on the observation that in traditional programming much time is wasted by continuously adapting algorithms to ever changing data structures. This becomes a significant problem in application areas where algorithms play a crucial role. Here, algorithms should be considered as independent building blocks rather than being dependent on specific data structures. Therefore, generic programming puts its main emphasis on *separating algorithms from data structures by means of abstract interfaces* between them.[2] Since algorithms indeed play a central role in image analysis and computational topology, generic programming suggests itself as a suitable software design method in these fields.

The algorithm centered approach of generic programming consists of 5 steps:

1. Analyze the requirements of algorithms in the intended field. Ensure that each algorithm imposes *minimal* requirements on the underlying data structures.
2. Organize the set of requirements into *concepts*. A concept contains a minimal set of functionality that is typically needed together.
3. Define abstract interfaces for all concepts. Map the definitions onto appropriate generic constructs of the target programming language (e.g. templates in C++).
4. Implement the interfaces on top of suitable data structures. A data structure may implement several concepts, and each concept may be realized by several data structures.
5. Implement algorithms in terms of abstract concepts (interfaces). Thus, algorithms can run on top of *any* data structure that implements the concepts.

The currently best example for this approach is the design of the *Standard Template Library* [1] which forms a major part of the C++ standard library. This library deals with fundamental data structures (list, arrays, trees etc.) and algorithms (sorting, searching etc.). On the basis of the steps sketched above, Stepanov and Musser arrived at the *iterator* concept as their fundamental interface abstraction. Because of its importance for our own discussion, we will describe iterators in some detail.

---

[2] This extents the traditional theory of abstract data types and object-oriented programming, where abstract interfaces have only been defined for data structures and objects.

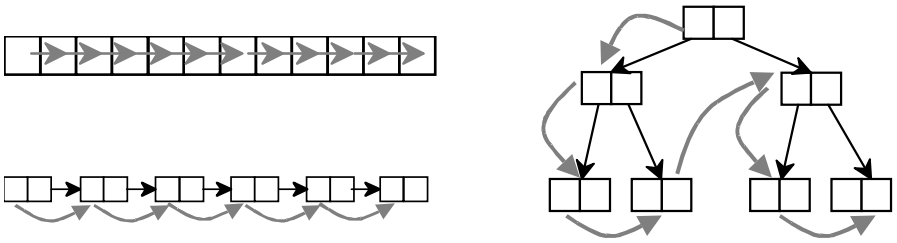

Fig. 3. Effect of a call `++iterator` on different data structures. top left: array – the iterator is moved to the next memory location; bottom left: linked list – the iterator follows the link to the next item; right: tree – the iterator realizes a pre-order traversal (black arrows denote links in the data structure, gray ones show the path of the iterator)

Iterators

Algorithm analysis revealed that many algorithms need to sequentially access the individual data items in a container. Although there exists a wealth of different sequences (such as arrays, deques, and lists), most algorithms use them in very simple ways. For example, they access one data item, process it, and then proceed to the next or previous item. It was Stepanov and Musser's key insight that the commonalties can be reinforced and the differences be encapsulated by means of *iterators*. In the example, a bi-directional iterator would be required which has the following capabilities[3]:

1. It points to a specific item in a specific container, and allows reading and possibly writing the data of that item. In C++, this is realized by the dereferencing operator: `item = *iterator`
2. It can be moved to the next and previous items. In C++, this is done using the increment and decrement operators: `++iterator` and `--iterator`
3. Two iterators can be compared to find out whether they point to the same item. This is realized by the quality operator: `iterator == end_iterator`.
   (This is used to determine the range of iteration: each data structure provides a pair of iterators that point to the start and one-past-the-end of the sequence respectively. The former iterator is incremented until it compares equal to the latter one, which signals the end of the sequence.)

Key to the efficiency of algorithms using these iterators is the additional requirement the all four operations must execute in constant time. Iterators meeting these requirements can be implemented for many different data structures, as shown in fig. 3. In case of an array, the iterator points to a memory location, and the increment operation moves it to the next memory location. In case of a linked list, a pointer to the next item is stored together with each item, the increment operator follows this link. For a tree, one can implement different iterators that realize pre-, post-, and in-order traversals respectively, by following the edges of the tree in the appropriate order.

---

[3] In addition, Stepanov and Musser define input, output, forward, and random access iterators. These will not be needed in this paper.

Algorithms are then defined so that they use only iterators, not the underlying data structures. By implementing algorithms as C++ *templates*, it is possible to leave the iterators' types open until the algorithm is actually applied to a concrete data structure. Only then will the compiler replace the formal (place-holder) type of the iterator with an actual type. For example, the signature of the `copy()`-algorithm which copies the items from the source sequence to the target sequence, looks like this:

```
template <class SourceIterator, class TargetIterator>
void copy(SourceIterator src_begin, SourceIterator src_end,
          TargetIterator target_begin);
```

For further information about the design of the STL and generic programming in general, [1] is recommended.

In case of cyclic data structures, the concept of an iterator (which has a first and last element) is replaced by the concept of a *circulator* [5]. A circulator behaves like an iterator, with the exception that there is no special last element where further incrementing is forbidden. Moreover, a circulator needs an additional function `iterator.isSingular()` which returns `true` iff the cyclic sequence is empty. The circulator concept will be important in the context of cell complexes as their orbits form cyclic sequences.

4 Generic Concepts for Cell Complexes

The development of generic concepts for cell complexes must start with an analysis of algorithm requirements. This has been done by several researchers, e.g. [2, 5, 7]. To illustrate this process I will present a simple segmentation algorithm. A more extensive discussion of generic topological algorithms is beyond the scope of this paper and will be given elsewhere.

The algorithm "Connected Components Segmentation" groups 2-cells together according to a predicate are_similar(2-cell, 2-cell) that is applied to all adjacent 2-cells (2-cells are adjacent if they are bounded by a common 1-cell). The exact nature of the predicate is application dependent and thus left unspecified here. A unique label is assigned to the 2-cells in each connected region of "similar" cells. After labeling all 2-cells according to similarity, labels are assigned to 0- and 1-cells according to the following rule: if a 0- or 1-cell bounds only 2-cells with the same label, it is assigned this same label, otherwise it is assigned a special BORDER_LABEL. It should be noted that this algorithm essentially creates block complexes according to definition 4, with the exception that 2-blocks need not be simply connected but may have holes.

In Table 1 we give the pseudo-code of the algorithm on the left and corresponding requirements on the right. Although this simple algorithm does not exploit the entire functionality of a cell complex, it illustrates nicely how sets of abstract requirements are formulated: the algorithm can be applied to any data structure that supports the operations listed on the right. In order to maximize the scope of the algorithm these operations should represent *minimal* requirements. For example, the algorithm doesn't care in which order the 1-cells incident to a 0-cell are listed. So it would be wrong to require those cells to be listed in a particular ordered (say, in orbit order), because this would needlessly restrict the applicability of the algorithm.

Pseudo-code	Data structure requirements
// find connected 2-cells of single component	
define recurse_component(2-cell c):	
for each adjacent 2-cell ac **of** c **do**:	// list adjacent 2-cells of given 2-cell
if label(ac) != NO_LABEL:	// read integer attribute of 2-cell
continue *// ac already processed*	
else if are_similar(c, ac):	// evaluate predicate
label(ac) = label(c)	// assign integer attribute to 2-cell
recurse_component(ac)	
// find all connected components and	
// label cells accordingly	
define cc_segmentation(cell_complex):	
for each cell c **in** cell_complex **do**:	// find all cells in a cell complex
label(c) = NO_LABEL	// associate initial attribute with cells
maxlabel = NO_LABEL + 1	
// phase 1: 2-cell labeling	
for each 2-cell c2 **in** cell_complex **do**:	// list all 2-cells of a cell complex
if label(c2) != NO_LABEL:	// read integer attribute of cell
continue *// c2 already processed*	
else:	
label(c2) = maxlabel	// assign integer attribute to cell
maxlabel = maxlabel + 1	
recurse_component(c2)	
// phase 2: 1-cell labeling	
for each 1-cell c1 **in** cell_complex **do**:	// list all 1-cells of a cell complex
label1= label(firstBounded2Cell(c1))	// access the two 2-cells bounded ...
label2= label(secondBounded2Cell(c1))	// ... by the current 1-cell
if leftLabel == rightLabel:	
label(c1) = leftLabel	// assign integer attribute to cell
else:	
label(c1) = BORDER_LABEL	// assign integer attribute to cell
// phase 3: 0-cell labeling	
for each 0-cell c0 **in** cell_complex **do**:	// list all 0-cells of a cell complex
for each incident 1-cell ic **of** c0 **do**:	// list 1-cells bounded by current 0-cell
if label(ic) == BORDER_LABEL:	
label(c0) = BORDER_LABEL	// assign integer attribute to cell
break	
else:	
label(c0) = label(ic)	// assign integer attribute to cell

Table 1. Connected components segmentation algorithm

Of course, it is easy to keep requirements abstract in pseudo-code. But traditional programming techniques, especially procedural programming, do not allow to carry over the abstraction into executable code: they force the programmer to fix a particular data structure so that the algorithm's potential flexibility cannot be realized in practice. This is not the case in generic programming and its realization by C++ templates – algorithms are independent of concrete data structures so that requirements can be translated into executable code *without losing abstraction*.

Analysis of topological algorithms revealed the following recurring requirements:

Combinatorial Requirements:

Analysis of the combinatorial structure

- list all k-cells of a cell complex
- list all incident cells for a given cell (in any order or in the cyclic order defined by definition 2)

Modification of the combinatorial structure

- elementary transformations between cell complexes

Data Association Requirements:

Assignment of data to cells and group of cells

- temporary data (e.g. label of the connected component each cell belongs to)
- geometric data (coordinates, model parameters)
- application data (requirements vary extremely from application to application, e.g. regions statistics, object interpretations, textures etc.)

It should be noted that this analysis suggests a strict separation between topological and geometrical properties. For a given topological structure, there are many different ways to represent its geometry – for example by using polygons, implicit equations, or sets of pixels/voxels. Separation of concerns therefore requires to treat topology and geometry as independent dimensions of abstraction. We will not explicitly deal with geometry in this paper, but concentrate on topology. However, it should be pointed out that representation of geometry is a chief application of *data association with cells* as described in the next subsection.

Data Association with the Cells

There are three basic possibilities to associate data with the cells of a cell complex:

Store Data in the Cell Data Structure: In addition to the information that encodes the combinatorial structure of the cell complex, application data can be stored directly in the cells. This data is retrieved by member functions of the cell objects. Templates are a great help in designing a cell complex that way.

Store Data outside the Cells: This technique requires to store a unique ID with each cell. This ID is used to retrieve additional application data from external data structures such as hash tables. This technique is more flexible than the first because the cell data structures need not be modified to add a new attribute, but it may be slower and requires more care from the user.

Calculate Data on Demand: Some data are redundant and need not be stored at all. If needed, these data can be calculated from other data that was stored. For example, the length of an edge can often be computed as the distance between the start and end node of the edge.

Now we want to hide these possibilities behind an interface, so that algorithms need not know how the data are stored. This is important, because otherwise all algorithms would have to be adapted when data must be stored differently. A suitable technique to achieve this is the so called *data accessor* [10].

A data accessor is a helper object that retrieves an attribute associated with a data item, given a handle to the item. It is similar to a member function in that it hides how the attribute is stored, but it is more flexible because it is independent of a particular data structure. As an example, let's consider data accessors for the length of an edge according to each of the three possibilities above:

```
struct LengthStoredInCellAccessor {
    float get(EdgeHandle edge) {
        // call member function to retrieve length attribute
        return edge.length();
    }
};

struct LengthStoredExternallyAccessor {
    Hashtable edge_length;

    float get(EdgeHandle edge) {
        // use edge ID to retrieve length from hash table
        return edge_length[edge.ID()];
    }
};

struct LengthCalculatedAccessor {
    float get(EdgeHandle edge) {
        // find differenc vector between start and end node
        Vector diff = edge.startNode().coordinate() -
                      edge.endNode().coordinate();
        // calculate length as magnitude of difference vector
        return diff.magnitude();
    }
};
```

The important point about these accessors is that they are used uniformly by algorithms: algorithms uniformly call `lengthAccessor.get(edgeHandle)`, without knowing which accessor is currently used. Iterators and circulators will play the role of cell handles, as they always refer to a "current" element. See [10] for more details.

5 Generic Concepts for Analyzing the Combinatorial Structure

Concepts for analyzing the combinatorial structure are the most interesting part of the cell complex interface. We will define this interface on the basis of the combinatorial map's darts and orbits. This doesn't imply that the interface is not applicable to other topological structures, it just simplifies the definitions.

Iterators and circulators will form the central parts of the interface. First, we need iterators that simply list all cells of a given dimension:

k-Cell Iterator: A k-cell iterator (k=0, 1, 2) iterates over all k-cells in a cell complex, in an arbitrary but fixed order.

Second, we need iterators for the different orbits that determine the combinatorial structure. Since orbits are cyclic, we actually need circulators here:

Orbit Circulator: An orbit circulator lists the darts in one of the orbits (σ, α, φ). It is bi-directional, i.e. it can step to the successor and predecessor of a given dart.

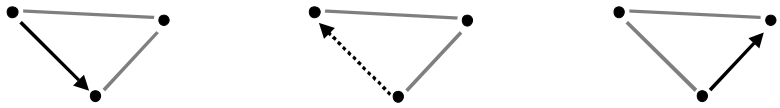

Fig. 4. Incrementing a φ-circulator which goes around the interior face of the graph. left: initial position; center: intermediate position after α-iteration of embedded circulator; right: final position after σ-iteration of embedded circulator

For all iterators and circulators, we will use the usual `++i` and `--i` notation to step forward and backward. The only exception is the α-circulator: since each α-orbit contains only two darts, we use the call `i.jumpToOpposite()` to switch between the two elements of an α-orbit, i.e. between the two ends of an edge. Circulators can thus provide all three operations simultaneously, so that σ- and φ-circulators automatically play the role of α-circulators as well. This will simplify many algorithms.

Now we are already able to give a completely generic implementation of the φ-circulator which encodes the orbit of darts around a face: since the φ-permutation can be composed from the other two, we only need a σ-circulator (which always is an α-circulator as well) to build it. Thus, we store a σ-circulator within the φ-circulator, and all φ-operations are implemented in terms of the appropriate σ and α-operations:[4]

```
template <class SigmaCirculator>
class PhiCirculator {
    SigmaCircultor sigma;
  public:

    // pass the embedded σ-circulator
    PhiCirculator(SigmaCirculator contained)
    : sigma(contained)
    {}

    // implement the φ-increment as σ^-1α of the embedded circulator
    void operator++() {
       contained.jumpToOpposite(); // one α iteration
       --contained;                // one backward σ iteration
    }

    // implement the φ-decrement as α^-1σ of the embedded circulator
    void operator++() {
       ++contained;                // one σ iteration
       contained.jumpToOpposite(); // one α^-1 iteration
    }
    ... // more functions
};
```

This code works regardless of what type the contained circulator has and what kind of cell complex we are using. Figure 4 illustrates a single φ-increment.

---

[4] We use C++ for the implementation examples on two reasons. First, at present there is no language-independent abstract notation for generic components. Second, it demonstrates that the concepts presented can be and actually have been implemented in practice.

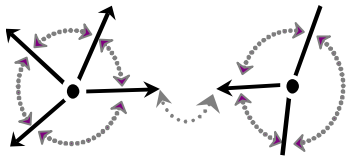

Fig. 5. Half-edges and their linking into orbits

Implementation of the σ-Circulator for a Half-Edge Data Structure

Now, we want to explain the implementation of the σ-circulator for different kinds of cell complexes. We start with a half-edge data structure, because this is the most direct implementation of a combinatorial map. The central concept of the half-edge data structure is the half-edge or dart, which explicitly stores pointers to its successors and predecessors in both the σ- and α-orbits. A typical implementation of a half-edge looks like this (see also figure 5):

```
struct HalfEdge {
    HalfEdge * sigma_next, * sigma_previous, * alpha_other;
    ... // more data not relevant here
};
```

Since the half-edge stores all necessary navigation information, the σ-circulator can simply store a pointer to the current half-edge. Upon a call to the increment operator it will look up the appropriate neighboring half-edge and replace the stored half-edge with this one. An initial half-edge is passed to the circulator in the constructor:

```
class HalfEdgeSigmaCirculator {
    HalfEdge * current;
  public:
    HalfEdgeSigmaCirculator(HalfEdge * initial)
    : current(initial)
    {}
    void operator++() {
        // goto successor in σ orbit
        current = current->sigma_next;
     };
     void operator--() {
        // goto predecessor in σ orbit
        current = current->sigma_previous;
     };
     void jumpToOpposite() {
        // goto successor (= predecessor) in α orbit
        current = current->alpha_other;
     };
     ... // more functions
};
```

This circulator gives us access to the *complete* combinatorial structure of the map: any dart, any orbit, and thus any cell can be reached by an appropriate operation sequence.

Fig. 6. left: four possible directions of the `KhalimskySigmaCirculator` in the σ-orbit; right: two possible directions of the circulator in the α-orbit

Implementation of σ-Circulator for Khalimsky's Topology

Since Khalimsky's topology is defined on a rectangular grid, we can use the grid coordinates to uniquely identify each cell. Note that coordinates are only used in the implementation and not exposed through the circulator interface. We assume that points whose coordinates are both odd are considered 0-cells. Then, points with two even coordinates are 2-cells, and the remaining points 1-cells. A dart is an oriented 1-cell. Hence, we add the attribute "direction" to distinguish the two darts of a 1-cell. Since there are four possible directions in a rectangular grid, we will denote directions by the integers 0...3 to encode "right", "up", "left", and "down" respectively.

Thus, a dart is uniquely determined by its direction and the coordinate of its start node. The σ-orbit around that node is obtained by incrementing the direction modulo 4. Similarly, the opposite dart in the α-orbit is found by moving the dart's coordinate two pixels in its direction (namely to the end node of the corresponding 1-cell) and reversing the direction. Figure 6 shows the possible positions of the circulator in the σ- and α-orbits. This leads to the following code for the σ-circulator:

```
class KhalimskySigmaCirculator {
    int x, y, direction;
  public:
    // init with a coordinate and a direction
    KhalimskySigmaCirculator(int ix, int iy, int idir)
    : x(ix), y(iy), direction(idir)
    {}

    void operator++() {
        // next direction counter-clockwise
        direction = (direction + 1) % 4;
    }

    void operator--() {
        // next direction clockwise
        direction = (direction + 3) % 4;
    }

    void jumpToOpposite() {
        static int xStep[] = { 2, 0, -2, 0};
        static int yStep[] = { 0, -2, 0, 2};

        x += xStep[direction];            // go to the other end of
        y += yStep[direction];            // the current 1-cell
        direction = (direction + 2) % 4;  // reverse direction
    }
    ... // more functions
};
```

Fig. 7. left: a block complex (cf. figure 2 right); right: possible positions of the block complex σ-circulator around the black 0-cell – only two 1-cells (black) belong to a 1-block, the other two (gray) are ignored because they belong to a 2-block

Implementation of the σ-Circulator for a Block Complex

Finally, we want to demonstrate how the σ-circulator can be implemented for a block complex. A block complex differs from the two cell complex implementations described so far by the fact that blocks are made up of several cells of an underlying cell complex. Therefore, we can use the circulators of the underlying cell complex to implement the block complex's circulators. Since the circulators have identical interfaces for all cell complex types, the block complex circulators can be written completely generically, independent of the type of the underlying cell complex.

In order to analyze the structure of a block complex, we need some means to tell which kind of block each cell belongs to. Therefore, we require a data accessor that returns the dimension of the block for each 0- and 1-cell (2-cells always belong to 2-blocks). For each underlying cell complex, this accessor is the only externally visible class that must be implemented in order to create a block complex:

```
struct BlockDimensionAccessor {
    int dimensionOfEdge(SigmaCirculator) {
        ... // suitable implementation for underlying cell complex
    }
    int dimensionOfNode(SigmaCirculator) {
        ... // suitable implementation for underlying cell complex
    }
};
```

The second function will return the dimension for the *start node* of the corresponding dart. Then the σ-orbit of a given 0-block is identical to the σ-orbit of the underlying 0-cell, except that all 1-cells belonging to a 2-block are ignored (see figure 7).

The navigation in an α-orbit is slightly more difficult: given one end of a 1-block, we must find the 0-cell at the other end of this 1-block. This is essentially a *contour following* algorithm: check if the *end* node of the current dart is a 0-block. If yes, we have found the other end of the 1-block. Otherwise jump to the next dart in the 1-block and repeat (see figure 8). Therefore, the complexity of an α-increment in a block complex is proportional to the length of the path that constitutes the current 1-block. The following circulator implementation realizes this algorithm (in `jumpToOpposite()`), as well as the selection of darts for the σ-orbit (in `operator++()` and `operator--()`):

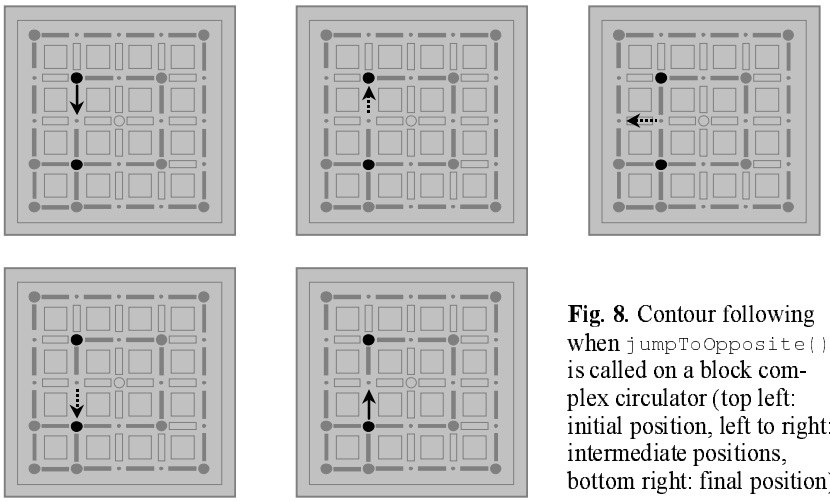

Fig. 8. Contour following when `jumpToOpposite()` is called on a block complex circulator (top left: initial position, left to right: intermediate positions, bottom right: final position)

```
template <class SigmaCirculator, class BlockDimensions>
class BlockComplexSigmaCirculator {
    SigmaCirculator embeddedSigma;
    BlockDimensions blockDimension;
  public:

    // embed a circulator for the underlying cell cimplex and a
    // block dimension accessor
    BlockComplexSigmaCirculator(SigmaCirculator embedded,
                                BlockDimensions dims)
    : embeddedSigma(embedded), blockDimension(dims)
    {}

    void operator++() {
        do {
            // increment embedded circulator ...
            ++embeddedSigma;
        // ... but ignore 1-cells that are not in 1-blocks
        } while(blockDimension.dimensionOfEdge(embeddedSigma) != 1);
    }

    void jumpToOpposite() {
        // contour following:
        // goto the other end of the current 1-cell
        embeddedSigma.jumpToOpposite();
        // until we find the other end of the 1-block ...
        while(blockDimension.dimensionOfNode(embeddedSigma) != 0) {
            // ... follow the cell chain in the 1-block
            operator++();
            embeddedSigma.jumpToOpposite();
       }
    }
    ... // more functions
};
```

Application: Generic Implementation of Connected Components Segmentation

We can now translate the pseudo-code for connected components segmentation given in table 1 into actual C++ code. The correspondences between the two versions of the algorithm should be easy to see. In order to keep the algorithm independent of the data structures, the C++ function is declared as a template of a `CellComplex` data structure and a store for additional `CellData` (which holds the cells' labels here).

The abstract requirements on the data structure are translated into generic function calls as follows: The functions `fi = c.beginFaces()` and `cc.endFaces()` specify the range of an iterator that lists all 2-cells in the cell complex `cc`. `*fi` returns a unique handle that identifies the current 2-cell. Analogous functionality is provided for 1- and 0-cells. `pi = fi.phiCirculator()` returns a circulator for the φ-orbit associated with the given 2-cell. `pi.face()` gives again a handle to that 2-cell. `pi.opposite()` retrieves the other dart in `pi`'s α-orbit and creates a φ-circulator associated with that dart. Consequently, `pi.opposite().face()` is a 2-cell adjacent to the one given by `pi.face()`. `ei.firstBoundedFace()` and `ei.secondBoundedFace()` give the two 2-cells bounded by the current 1-cell (where `ei` is an 1-cell iterator and the assignment of "first" and "second" is arbitrary). Finally, `si = ni.sigmaCirculator()` creates a circulator for the σ-orbit of the current 0-cell referenced by node iterator `ni`, and `si.edge()` returns the handle to the edge the current dart of σ-circulator `si` is part of. It shall again be emphasized that these function calls work uniformly across all kinds of topological data structures defined in this paper.

```
template <class CellComplex, class CellData>
void
connectedComponentsSegmentation(CellComplex const & cc, CellData & cd)
{
  // initialise the cells' labels to NO_LABEL
  // 2-cells
  CellComplex::FaceIterator fi = cc.beginFaces();
  for(; fi != cc.endFaces(); ++fi) cd.label(*fi) = NO_LABEL;
  // 1-cells
  CellComplex::EdgeIterator ei = cc.beginEdges();
  for(; ei != cc.endEdges(); ++ei) cd.label(*ei) = NO_LABEL;
  // 0-cells
  CellComplex::NodeIterator ni = cc.beginNodes();
  for(; ni != cc.endNodes(); ++ni) cd.label(*ni) = NO_LABEL;

  // counter for the connected components
  int current_label = NO_LABEL + 1;
  // find all connected components of 2-cells
  for(fi = cc.beginFaces(); fi != cc.endFaces(); ++fi)
  {
    if(cd.label(*fi) != NO_LABEL)
      // 2-cells has already been processed in a prior call to
      // recurseConnectedComponent() -> ignore
      continue;

    cd.label(*fi) = current_label++; // start a new component
    // get the φ-circulator for the given 2-cell
    CellComplex::PhiCirculator pi = fi.phiCirculator();
    // recurse to find 2-cells belonging into current component
    recurseConnectedComponent(pi, cd);
  }
```

```
  // label the 1-cells according to the labels of the 2-cells they bound
  for(ei = cc.beginEdges(); ei != cc.endEdges(); ++ei)
  {
    int label1 = cd.label(ei.firstBoundedFace());
    int label2 = cd.label(ei.secondBoundedFace());
    if(label1 == label2)
      // if the two 2-cells belong to the same component, the 1-cell
      // also belongs to that component
      cd.label(*ei) = label1;
    else
      // otherwise, the 1-cell is a border cell
      cd.label(*ei) = BORDER_LABEL;
  }

  // label the 0-cells according to the labels of the 1-cells they bound
  for(ni = cc.beginNodes(); ni != cc.endNodes(); ++ni)
  {
    // get circulator for σ-orbit of current 0-cell
    CellComplex::SigmaCirculator si = ni.sigmaCirculator();

    int label = cd.label(si.edge());   // remember label of first 1-cell
    do {
      if(cd.label(si.edge()) == BORDER_LABEL)
      {
        // if any of the 1-cells bound by the current 0-cell is
        // a border cell the 0-cell is also a border cell
        label = BORDER_LABEL; break;
      }
    } while(++si != ni.sigmaCirculator());
    cd.label(*ni) = label;  // assign label to 0-cell
  }
}

// recursively mark all 2-cells of a component with the same label
template <class PhiCirculator, class CellComplex, class CellData>
void recurseConnectedComponent(PhiCirculator pi, CellData & cd)
{
  int current_label = cd.label(pi.face()); // remember current label

  do {
    // get φ-circulator for adjacent 2-cell
    PhiCirculator opposite_pi = pi.opposite();

    if(cd.label(opposite_pi.face())== NO_LABEL &&
       are_similar(pi.face(), opposite_pi.face()))
    {
      // if the adjacent 2-cell is similar and not yet assigned,
      // label it and recurse (are_similar() is not discussed here)
      cd.label(opposite_pi.face()) = current_label;
      recurseConnectedComponent(opposite_pi, cd)
    }
  }
  while(++pi != fi.phiCirculator());
}
```

The above algorithm is very efficient: since all iterator/circulator functions execute in constant time (except in a block complex), the overall algorithm has complexity $O(N\sigma + E + F\varphi)$, where N, E, and F denote the number of 0-, 1-, and 2-cells, and σ

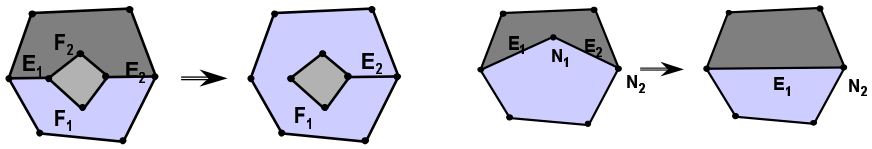

Fig. 9. left: merging of two 2-cells by removing a common 1-cell; right: merging of two 1-cells by removing the common 0-cell

and φ are the average lengths of the σ- and φ-orbits respectively. This is much more efficient than the approach of Dufourd and Puitg [3], where the functions to access the next dart in an orbit take linear time in the size of the cell complex (i.e. $O(N + E + F)$), which makes the overall algorithm roughly $O((N + E + F)^2)$.

Elementary Transformations Between Cell Complexes

The question of a uniform interface for cell complex *transformations* (insertion and removal of cells) is much more difficult to answer than that for the operations that merely navigate on an existing cell complex. This has several reasons:

1. Different applications have quite different requirements. For example, in numeric analysis (FEM) and computer graphics, mesh refinement is a key operation. It is often done by replacing a cell with a predefined refinement template, so that regularity is preserved. In contrast, in image analysis we are more interested in merging neighboring cells in order to build irregular image pyramids.
2. Different cell complex implementations support different sets of operations. For example, a half-edge data structure can essentially support any conceivable transformation. In contrast, a Khalimsky topology cannot be transformed at all. Block complexes are between the extremes: we can always merge blocks, but we can split only blocks that consist of more than one cell.

Therefore, I will only briefly show which elementary transform have emerged as particularly useful in the field of image analysis. The term "elementary" here refers to transformations that only involve few cells, so that only a very localized area of the cell complex is changed and the operation can be implemented very efficiently. More complex transformations can be build on top of the elementary ones.

The elementary transformations are often called *Euler operators* [11] because, on a planar cell complex, they must respect Euler's equation $v - e + f = 2$ (with v, e, and f denoting the number of nodes, edges, and faces respectively). In principle, two Euler operators and their inverses are sufficient to span the space of planar cell complexes (if we don't allow "holes" in faces – otherwise we will need three):

Merge Two 2-Cells: If two 2-cells are bound by a common 1-cell, we may remove that 1-cell and merge the 2-cells.

Merge Two 1-Cells: If a 0-cell bounds exactly two distinct 1-cells, we may remove the 0-cell and merge the 1-cells.

Figure 9 shows examples for either operation. By applying them in a suitable order, one can reduce any cell complex to a single node located in the infinite (exterior) face. The reverse operations are used to refine a cell complex. Complex operations are composed from the elementary ones. Sometimes this leads to very inefficient imple-

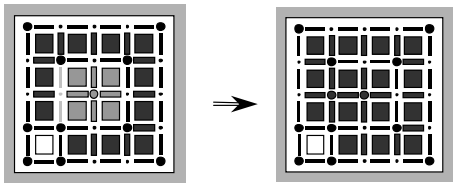

Fig. 10. merging of two 2-blocks in a block complex: when the 1-block marked gray is removed, the cells belonging to the surviving 2-block have to relabeled (in dark gray).

mentations of complex operations. Then one may add further specialized operations, but for many applications this is not necessary.
It is important to note that a direct implementation of the elementary operations is only possible if we can actually remove cells from the cell complex data structure. In case of a block complex, this is not the case. But we can still implement the Euler operators: instead of removing cells, we assign new block dimensions and labels to the cells according to the modified block structure, as figure 10 illustrates.

It should be noted that the raw Euler operators do not update the non-topological data associated to the cells involved (e.g. geometrical or statistical information). If such data are present, it is necessary to augment the Euler operators with additional code that handles the update. For example, when two 2-cells are merged, the average gray level of the combined 2-cell must be calculated from the statistics of the original cells. This can be done by wrapping the raw Euler operators into new operators (with the same interface) that carry out additional operations before and/or after calling the original Euler operator. This technique is also known as the *decorator pattern* [4]. Due to the flexibility of generic programming, templated algorithms need not be modified in order to support this: since the augmented operators still conform to the uniform interface definition, they can transparently be called instead of the raw operators, without the algorithms knowing the difference.

6 Conclusions

This article has shown that generic concepts can make the use of planar cell complexes in everyday programming much more convenient: interface concepts such as iterators, circulators, and data accessors shield algorithms from the particulars of the underlying data structure. The separation between algorithms and data structures is very important, because most of the know-how of image analysis and other fields using cell complexes lies in algorithms rather than data structures. This means that it is usually quite difficult and error prone to change existing algorithm implementations whenever the underlying data structures change. It is much easier to write some mediating iterators and accessors and leave the algorithms untouched. The resulting solutions combine the speed of C++ with the simplicity of the functional approach of Dufourd and Puitg [3].

Connected components segmentation is a good case in point. But the advantages of generic programming become even more apparent when we apply a segmentation

algorithm repeatedly and in turn with Euler operations in order to generate an irregular segmentation pyramid: Initially, we interpret the entire image as a Khalimsky plane, where the pixels are considered 2-cells and the appropriate 0- and 1-cells are inserted between them. Cells are labeled by means of a segmentation algorithm to create the first level of the segmentation pyramid (usually, this is an oversegmentation). The components (blocks) at this level are then merged into single cells by a suitable sequence of Euler operators. This process is repeated to get ever coarser pyramid levels, until the final segmentation is satisfying. In order to optimize algorithm performance, it is a good idea to use a block complex on top of the Khalimsky plane for the first few levels, and switch to a half-edge data structure when regions become larger. By using the interfaces described here, we can implement segmentation and merging algorithms independently of the data structures used. Choosing the appropriate kind of cell complex at every pyramid level thus becomes very easy.

Of course, there is still much to be done. In particular, more cell complex algorithms should be implemented according to the generic paradigm to realize the advantages claimed here and gain new insights for further improvement. The interaction between topology and geometry should be investigated more deeply. A wider choice of specialized Euler operators and extension to 3D would also be desirable. Nevertheless, the approach is already working and shows promising results, with good performance (a few seconds) in segmenting cell complexes with thousands of cells.

7 References

[1] M. Austern: *"Generic Programming and the STL"*, Reading: Addison-Wesley, 1998

[2] G. Berti: *"Generic Software Components for Scientific Computing"*, PhD thesis, Fakultät für Mathematik, Naturwissenschaften und Informatik, Brandenburgische Technische Universität Cottbus, 2000

[3] J.-F. Dufourd, F. Puitg: *"Functional specification and prototyping with oriented combinatorial maps"*, Computational Geometry 16 (2000) 129-156

[4] E. Gamma, R. Helm, R. Johnson, J. Vlissides: *"Design Patterns"*, Addison-Wesley, 1994

[5] L. Kettner: *"Designing a Data Structure for Polyhedral Surfaces"*, Proc. 14th ACM Symp. on Computational Geometry, New York: ACM Press, 1998

[6] E. Khalimsky, R. Kopperman, P. Meyer: *"Computer Graphics and Connected Topologies on Finite Ordered Sets"*, J. Topology and its Applications, vol. 36, pp. 1-27, 1990

[7] U. Köthe: *"Generische Programmierung für die Bildverarbeitung"*, PhD thesis, Computer Science Department, University of Hamburg, 2000 (in German)

[8] V. Kovalevsky: *"Finite Topology as Applied to Image Analysis"*, Computer Vision, Graphics, and Image Processing, 46(2), pp. 141-161, 1989

[9] V. Kovalevsky: *"Computergestützte Untersuchung topologischer Eigenschaften mehrdimensionaler Räume"*, Preprints CS-03-00, Computer Science Department, University of Rostock, 2000 (in German)

[10] D. Kühl, K. Weihe: *"Data Access Templates"*, C++ Report Magazine, July/August 1997

[11] M. Mäntylä: *"An Introduction to Solid Modeling"*, Computer Science Press, 1988

[12] D. Musser , A. Stepanov: *"Algorithm-Oriented Generic Libraries"*, Software – Practice and Experience, 24(7), 623-642, 1994

[13] *"TargetJr Documentation"*, 1996-2000 (http://www.targetjr.org/)