Designing and implementing a general purpose halfedge data structure

Hervé Brönnimann[1]

[1] Polytechnic University, Brooklyn NY 11201, USA
hbr@photon.poly.edu

1 Introduction

Halfedge data structures (HDS) are fundamental in representing combinatorial geometric structures, useful for representing any planar structures such as plane graphs and planar maps, polyhedral surfaces and boundary representations (BREPs), two-dimensional views of a three dimensional scene, etc. Many variants have been proposed in the literature, starting with the winged-edge data structure of Baumgart[2], the DCEL of [15, 9], the quad-edge data structure [11], the halfedge data structure [18, 12, and refs. therein]. They have been proposed in various frameworks (references too many to give here):

– Plane structures: including planar maps for GIS, 2D Boolean modeling, 2D graphics, scientific computations, computer vision. The requirements on HDS are that that some edges may be infinite (e.g., Voronoi diagrams), or border edges (e.g., for bounded polygonal domains), it may include holes in the facets (planar maps), and that if so, one of the connected boundary cycle is distinguished as the outer boundary (the others are inner holes).
– Boundary representation of three-dimensional solids: including Brep representation, solid modeling, polyhedral surfaces, 3D graphics. The requirements here vary slightly: holes may still be allowed, but there is no need to distinguish an outer boundary, infinite edges are not always useful but border edges might need to be allowed.
– Planar structures encountered in higher dimensional structures: even though the data structure itself may be higher dimensional, we might want to interpret some two-dimensional substructure by using a HDS. Examples include the polygonal facets of a 3D model, or the local structure in a neighborhood of a vertex in a 3D subdivision, or the two-dimensional view of a three-dimensional scene.
– Special structures such as triangulations or simplicial complexes: in these structures, the storage is facet-based. They are usually easier to extend to higher dimensions, and a systematic presentation is given in [4].

All the implementations we are aware of, including those surveyed above, capture a single variant of HDS, with the notable exception of the design of halfedge data structure in CGAL presented in [12] which still limits itself to facet-based variants and does not allow holes in facets, for instance, but provides some variability (forward/backward, with or without vertices, facets, and their corresponding links, see below). This design was done for visualization of 3D polyhedral surfaces, and can be reused in several situations.

Virtually everybody who has programmed a DCEL or halfedge structure knows how difficult it is to debug it and get right. Often, bugs arise when adding or removing features for reuse in a different project. Hence, the problem we deal with here is to present a design which can express as many as possible of the variants of HDS that have been proposed in the literature, in order to design a *single set* of generic algorithms that can operate on *any* of them. The goals for our halfedge data structure design are similar to those presented in [12] :

- genericity: our specifications need to adapt to many existing structures, regardless of their internal representation. This makes it possible to express *generic algorithms* on them. Our goal is to capture all the features mentioned above. genericity implies that if features are not required, but used nevertheless (perhaps because they are required by another algorithm to be applied subsequently), the algorithm should adapt gracefully and maintain those features as well.
- power of expression: the various restrictions on HDS models in the CGAL project led to the existence of three distinct data structures, one for polyhedral surfaces [12], one for planar maps [10], and another one for triangulations [3]. By contrast, we want to express all these structures using a single framework, and have a single set of algorithms to deal with these structures.
- efficiency: we do not want to sacrifice efficiency for flexibility. This entails not maintaining or interacting with unused features of a HDS, and putting minimum requirements for the algorithms manipulating an HDS. Efficiency can be achieved by using C++ templates (static binding) and compiler optimization. Also, attention needs to be paid to issues of locality of reference and memory layout.
- ease-of-use: attaching information to the vertices, edges, or facets should be easy, as well as reuse of the existing component. Also the interface needs to be uniform and easily learnable (somewhat standard). We reuse and extend the C++ STL framework of concepts and models, also know as generic programming [1], to the case of this pointer-based data structure. See also the Boost Graph Library [13] for similar treatment of graph algorithms.

This paper extends the excellent study presented in [12] for the CGAL library, by providing a generic and all-purpose design for the entire family of halfedge data structures and their variants. We validate the approach in a C++ template library, the HDSTL, which is described below.

The paper is organized as follows. In the next section, we present the unified framework for working with and describing variants of HDS. Then we describe a set of generic algorithms that apply to these structures. We introduce the HDSTL, a small template library (less than 5000 lines of code) which provides a wide range of models, which we evaluate both individually and used in a practical algorithm. We conclude by evaluating how this design meets the goals expressed above.

2 Concepts

Halfedge data structures have a high degree of variability. We follow the commonality/variability analysis framework of Coplien [8]. Briefly speaking, they may allow several representations (vertices, facet, or no none), as well as holes in facets, infinite edges (incident to a single vertex), boundary edges (incident to a single facet), etc. The representation itself may allow various access to the data structure, such as clockwise or counterclockwise traversal of a facet boundary, of a vertex cycle, access to the source or target vertex of a halfedge. Even the type of container for the components may be an array (linear storage), a list (linked storage), or other kinds of containers. But the commonality is also clear: the intent is to model 2-manifold topology, so every edge is incident to at most two facets, and in fact every halfedge is incident to at most one facet and has an opposite halfedge. Halfedges are ordered circularly along a facet boundary. Also every edge is incident to two vertices, and in fact every halfedge is incident to a unique source vertex.

In this description, a halfedge data structure is simply a structured set of pointers which satisfy some requirements. The names of those pointers and the requirements are grouped by concepts. This lets us easily describe what kind of a HDS is expected by an algorithm. The purpose of expressing concepts is to describe easily and systematically to what kind of a HDS a generic algorithm should apply. In the process of designing our concepts, we try and express invariants, and their consequences on the validity of the HDS. (The reader is referred for instance to [6] and more recent papers by these authors for a formal approach guaranteeing consistency.) We formulate our invariants such that validity can be checked in constant time per halfedge, vertex or facet.

2.1 Halfedge Data Structure (HDS)

The simplest concept requires only halfedges. The HDS gives access to the halfedge type, the halfedge handle type, and the halfedge container. The only operations guaranteed to be supported is to create a pair of opposite halfedges, to take the opposite of a halfedge, and to access the halfedges (via the interface given by the concept of Container in the C++ STL). We say that a pointer is ***valid*** if it points to a halfedge that belongs to the HDS halfedge container. The only requirement is that the opposite of a halfedge h is a valid halfedge g, and that *opposite(g)* is h itself. This is our first invariant I1.

I1. All the opposite pointers are valid, *opposite(h)*$\neq h$, and *opposite(opposite(h))*$=h$, for any halfedge h.

In this and in the sequel, the variable h denotes a halfedge *handle*, or descriptor, not the actual halfedge element which could be a much bigger type. In the sequel, h and g denote halfedge handles, v a vertex handle, and f a facet handle.

In order to guarantee Invariant I1, halfedges are created in opposite pairs. There is thus no *new_halfedge()* function, but *new_edge()* creates two halfedges.

Remark. It usually required in the literature that the handle is a pointer, which can be more generally captured by the C++ STL concept of Trivial Iterator, meaning the following expressions are valid: **h*, **h=x* (assignment, for mutable handles only), default

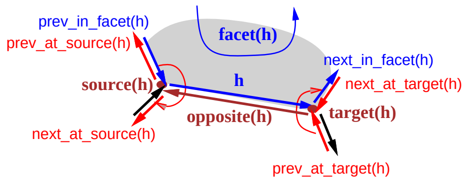


Fig. 1. The basic pointers in a HDS.

constructor, and *h->m* (equivalent to *(*h).m*). Because we want to attain the maximum generality, we would also like to allow handles to be simple descriptors, like an indices in a table. This precludes the use of notation like *h->opposite()* as used in e.g. CGAL. We therefore assume that the access to pointers is given by the HDS itself, or using C++ notation, by *hds.opposite(h)*. The cost of generality comes at somewhat clumsier notation.

2.2 Forward/Backward/Bidirectional HDS

In order to encode the order of the halfedges on the boundary of a facet, we must have access to the halfedge immediately preceding or succeeding each halfedge *h* on the facet cycle. In order to encode the order of the halfedges incident to a vertex, we must have access to the halfedge immediately preceding or succeeding each halfedge *h* on either the source or the target vertex cycle. We have now described the pointers that are involved in our halfedge data structure: in addition to the already described *opposite*(), the pointers that link halfedges together are *next_at_source*(), *next_at_target*(), *next_in_facet*(), *prev_at_source*(), *prev_at_target*(), *prev_in_facet*(). They are shown on Figure 1.

Note that all this information need not be stored. For instance, *next_at_source(h)* is the same as *next_in_facet(opposite(h))*, while *next_at_target(h)* is the same as *opposite(next_in_facet(h))*. In practice, since we always require access to the opposite of a halfedge, it suffices to store only one of *next_at_source*, *next_at_target*, or *next_in_facet* and one has access to all three!

We call a data structure in which one has access to all three pointers *next_at_source*, *next_at_target*, and *next_in_facet*, and that satisfy the invariant I2 below, a ***forward*** HDS.

I2. If a HDS is forward and satisfies invariant I1, then all the pointers *next_at_source*, *next_at_target*, and *next_in_facet* are valid, and for any halfedge *h*, we have *next_at_source(h)* = *next_in_facet(opposite(h))*, and *next_at_target(h)* = *opposite(next_in_facet(h))*.

Similarly, if one of the pointers *prev_at_source*, *prev_at_target*, or *prev_in_facet* is available, then all three are, and if they satisfy the invariant I3 below, the HDS is called a ***backward*** HDS.

I3. If a HDS is forward and satisfies invariant I1, then all the pointers *prev_at_source*, *prev_at_target*, and *prev_in_facet* are valid, and for any halfedge *h*, we have *prev_at_source(h)* = *opposite(prev_in_facet(h))*, and *prev_at_target(h)* = *prev_in_facet(opposite(h))*.

A data structure which is both forward and backward is called ***bidirectional***. We require that any HDS must provides access to either forward or backward pointers.[1] For a bidirectional HDS, we require the invariant:

I4. If a HDS is bidirectional, then *prev_at_source(next_at_source(h))=h, prev_at_target(next_at_target(h))=h* , and *prev_in_facet(next_in_facet(h))=h*, for any halfedge *h.*

2.3 Vertex-supporting and Facet-supporting HDS

The basic pointers at a halfedge give an axiomatic definition of vertices and facets. A source cycle is a non-empty range of halfedges $h_0 \ldots, h_k$ such that h_{i+1} =*hds.next_at_source(*h_i*)* for any halfedge h_i in this range and h_k =*hds.next_at_source(*h_1*)* if the HDS is forward, or such that h_i =*hds.prev_at_source(*h_{i+1}*)* for any halfedge h_i in this range and h_1 =*hds.prev_at_source(*h_k*)* if the HDS is backward. Likewise, a target cycle is defined similarly by using the *next_at_target* and *prev_at_target* pointers instead, and a (facet) boundary cycle by using *next_in_facet* and *prev_in_facet*. Two abstract vertices are ***adjacent*** if they contain at least a pair of opposite halfedges.

The important property of HDS is that each halfedge is incident to only one facet and has only two endpoints. Since the halfedge is oriented, the two vertices are therefore distinguished as the source and the target of the halfedge. We say that an HDS ***supports vertices*** if it provides the vertex type, the vertex handle type, access to the vertex container, as well as two pointers *source(h)* and *target(h)* for any halfedge *h*. Moreover, these pointers must satisfy the following invariants.

I5. If a HDS satisfies invariant I1 and supports vertices, then *source(g)=target(h)* and *target(g)=source(h),* for any pair of opposite halfedges *h* and *g*.

I6. If a HDS satisfies invariant I1 and supports vertices, then *source(next_in_facet(h))=target(h)* for any halfedge *h.*

Invariant I5 expresses that opposite halfedges have the same orientation, and Invariant I6 expresses that the halfedges on a boundary cycle are oriented consistently. Note that because of invariant I5, we need only store a pointer to the source or to the target of a halfedge. We may trade storage for runtime by storing both, but such a decision should be made carefully. More storage means the updates to the HDS take more time, therefore one needs to carefully evaluate whether the increased performance in following links is actually not offset by the loss of performance in setting and updating the pointers.

We express new invariants if vertices or facets are supported. Note that these invariants are checkable in linear time, and without any extra storage.

I7. If a HDS supports vertices, and satisfies invariants I1–I4, then *source(h)=source(g)* for any halfedges *h, g* that belong to the same source cycle, and *target(h)=target(g)* for any halfedges *h, g* that belong to the same target cycle

[1] Note that without this requirement, our data structure would consist of unrelated pairs of opposite edges. This is useless if vertices are not supported. If they are, it might be useful to treat such a structure like a graph, without any order on the halfedges adjacent to a given vertex. Still, it would be necessary for efficient processing to have some access to all the halfedges whose source (or target) is a given vertex. This access would enumerate the halfedges in a certain order. So it appears that the requirement is fulfilled after all.

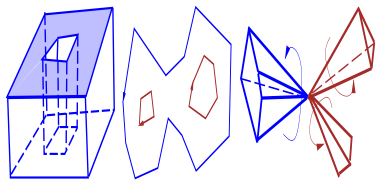


Fig. 2. An illustration of (a) facets with holes, (b) outer boundary, and (c) singular vertices.

I8. If a HDS supports facets, and satisfies invariants I1–I4, then *facet(h)=facet(g)* for any halfedges *h*, *g* that belong to the same boundary cycle.

2.4 Vertex and Facet Links

Even though our HDS may support vertices or facets, we may or may not want to allocate storage from each vertex of facet to remember one (perhaps all) the incidents halfedges. We say that a vertex-supporting HDS is ***source-linked*** if it provides a pointer *source_cycle(v)* to a halfedge whose source is the vertex *v*, and that it is ***target-linked*** if it provides a pointer *target_cycle(v)* to a halfedge whose source is the vertex *v*. A facet-supporting HDS is ***facet-linked*** if it provides a pointer *boundary_cycle(f)* to a halfedge on the boundary of any facet (in which case it must also provide the reverse access *facet(h)* to the facet which is incident to a given halfedge *h*). It is possible to envision use of both vertex- and facet-linked HDS, and non-linked HDS. The following invariants guarantee the validity of the HDS.

I9. If a HDS supports vertices, is source-linked, and satisfies Invariants I1–I7, then *source(source_cycle(v))=v* for every vertex *v*.

I10. If a HDS supports vertices, is target-linked, and satisfies Invariants I1–I7, then *target(target_cycle(v))=v* for every vertex *v*.

I11. If a HDS supports facets, is facet-linked, and satisfies Invariants I1–I6 and I8, then *facet(boundary_cycle(f))=f* for every facet *f*.

2.5 HDS with Holes in Facets and Singular Vertices

An HDS may or may not allow facets to have ***holes***. Not having holes means that each facet boundary consists of a single cycle; it also means that there is a one-to-one correspondence between facets and abstract facets. In a HDS supporting holes in facets, each facet is required to give access to a hole container.[2] This container may be global to the HDS, or contained in the facet itself. Each element of that container need only point to a single halfedge.

In a facet with holes, one of the cycles may be distinguished and called the ***outer boundary***; the other holes are the ***inner holes***. This is only meaningful for plane structure (see Figure 2(b)), where the outer boundary is distinguished by its orientation

[2] The container concept is defined in the C++ STL.

which differs from that of the inner holes. In Figure 2(a), for instance, the outer boundary is defined if we know that the facet is embedded in a plane, but there is no non-geometric way to define the outer boundary of the grayed facet: a topological inversion may bring the inside boundary cycle outside. (Note that when the incident facet is by convention to the left of the halfedge, the outer boundary is oriented counterclockwise, while the inner hole boundaries are oriented clockwise.)

In a connected HDS without holes, it is possible to reconstruct the facets without the facet links. In general, for an HDS with holes but which is not facet linked, it is impossible to reconstruct the facets as the information concerning which cycles belong to the same facet is lost, let alone which is the outer boundary. Therefore, if the HDS supports facets with holes, we require that it be facet-linked, and that it also provide the outer boundary and an iterator over the holes of any facet.

An HDS may also allow singular vertices. A vertex is ***singular*** if its corresponding set of adjacent halfedges consists of several cycles; it is the exact dual notion of hole in facet and has the same requirements concerning cycle container.

In a singular vertex, one of the cycles may be distinguished and called the ***outer cycle***, and the other cycles are called ***inner cycles***. They are depicted in Figure 2(c). Unlike the outer facet boundary, the possibility of inner cycles seems to only make sense for dual structures (see section on duality below) and may not be really useful, although it perhaps may be useful in triangulating 3D polyhedra (e.g. the algorithm of Chazelle and Palios keeps an outer boundary and may have inner vertex cycles when removing a dome). As with holes, if an HDS supports singular vertices, we require that it be vertex-linked and that it provide the outer cycle and an iterator over the inner cycles of any vertex.

2.6 Hole Links, Vertex Cycle Links, Infinite and Border Edges, Duality

By analogy with vertex and facet links, we call the HDS ***source-cycle-linked*** if it provides a pointer *source_cycle(h)* for each halfedge *h*, ***target-cycle-linked*** if it provides a pointer *target_cycle(h)* for each halfedge *h*, and ***hole-linked*** if it provides a pointer *boundary_cycle(h)* for each halfedge *h*.

An edge whose facet pointer has a singular value is called a ***border*** halfedge. Likewise, a halfedge whose source or target pointer has a singular value is called an ***infinite*** halfedge.

For lack of space, we cannot elaborate on these notions. Suffice it to note that with all these definitions, our set of concepts is closed under duality transformations which transform vertices into facets and vice-versa.

2.7 Memory Layout

In addition to the previous variabilities which have to do with the functionality of the data structure, and the model it represents, we offer some support for memory layouts. The only requirement we have made so far on the layout concerns the availability of halfedge, vertex and facet containers, as well as vertex cycle and hole containers if those are supported.

A ***mutable*** structure allows modification of its internal pointers, via functions such as *set_opposite(h,g)*, etc. These functions need only be supported if the corresponding pointer is accessed: a forward HDS is only required to provide *set_next_in_facet()*, *set_next_at_source()*, and *set_next_at_target()*. The reason we require separate functions for read or write access, is that a pointer may be accessed even though it cannot be set (if this pointer is non-relocatable, see next section).

There is considerable freedom in the implementation of an HDS. For instance, because of invariants I2 and I3, it is desirable to have pairs of opposite halfedges close in memory, so a possible way to do this as suggested by Kettner [12] is to store them contiguously. In this case the opposite pointer may be implicit and its storage may be reclaimed. The same trick could be used for any of the other pointers (such as *next_in_facet*, etc.).

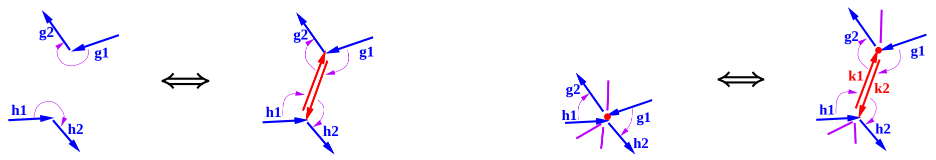
Another example is the CGAL triangulation, which can be captured in our framework as a HDS which is normalized by facets thus saving the storage for *all* bidirectional pointers. We call this layout a ***triangulated HDS***. There are thus six pointers only per triangular facet (three opposite and three source vertex pointers), which matches the CGAL triangulation data structure [3]. A restricted set of algorithms needs to be applied (*new_edge* replaced by *new_triangle*). This is unavoidable since a triangulated HDS cannot express all the possible states of a HDS. Note that some triangulation algorithms can maintain and work with a triangulated HDS (iterative and sweep algorithms) but others cannot because of their need of general HDS as intermediate representation (e.g., divide-and-conquer, see below).

Because of this freedom, we need to introduce one more concept: an HDS is ***halfedge-relocatable*** with respect to a given pointer if any two halfedge locations can be exchanged in the container, and the pointers to these halfedges updated, without affecting the validity of the HDS. An HDS which actually stores all its pointers is halfedge-relocatable, while the HDS given as example above, which store pairs of opposite halfedges contiguously, is not. Similar definitions can be made for ***vertex-*** and ***facet-relocatable*** HDS. These concepts are important in discussing normalization algorithms (see Section 3.3).

A halfedge-relocatable HDS provides a function *halfedge_relocate(h,g)* to relocate a halfedge pointed to by *h* to a position pointed to by *g* (whatever was at that position is then lost). It also provides a member function *halfedge_swap*(*h*,*g*) for swapping two halfedges in the containers without modifying the pointers in the HDS.

3 Generic Algorithms

The purpose of enunciating a collection of concepts is to describe precisely how the algorithms interact with a data structure. In the C++ STL, this interaction is achieved very elegantly through iterators: containers provide iterators, whose semantics can be modified by adapters, and algorithms operate on a range of iterators. In the HDSTL, the interaction takes place through a set of functors (such as *opposite, next_in_facet,* etc.), and whose arguments and return types are handles. Using this framework, we can express operators and algorithms on a HDS that specify exactly what the (compile-time) requirements are, and what the semantics of the algorithm are.



(a) *split_facet* (left-to-right) and *join_facets* (right-to-left). For instance, $h1, h2$ are successors in facet

(b) *split_vertex* (left-to-right) and *join_vertices* (right-to-left). For instance, $h1, g2$ are successors in facet.

Fig. 3. the meaning of the arguments *h1,h2,g1,g2* for the Euler operators.

3.1 Elementary Algorithms

In order to write generic algorithms, we need elementary manipulations of pointers. These are complicated by the fact that these pointers may or may not be supported, even the corresponding types may not exist. So we need “smart” functions which either provide the pointer or a default-constructed value if the pointer is not supported. The elementary functions come in several sets: the *get_...*, *set_...*, *copy_...*, *compare_...*, *create/destroy_...* functions.[3] In addition, in a valid HDS, the *next_...* and *prev_...* pointers may be computed by a reverse traversal of a (vertex or boundary) cycle, stopping before the cycle repeats. These are the *find_...* functions. For convenience, we provide *stitch_...* functions which set pairs of reverse pointers.

In [12], these functions exist and are encapsulated in a so-called decorator. We choose to provide them in the global scope since they are the primary access to the HDS. Their first argument is always an object of type HDS.

3.2 Euler and Other Combinatorial Operators

In addition to the elementary functions, most algorithms use the same set of high-level operations, called Euler operations since they preserve the Euler number of the structure ($v - e + f - i - 2(c + g)$, where v, e, f, i , c, are the numbers of vertices, edges, facets, inner holes, and connected components, and g is the genus of the map). The basic operation used by these operators is splice which breaks a vertex cycle by inserting a range of edges. Using *splice*, we provide an implementation of dual operator *join_facets* and *join_vertices* (delete a pair of opposite halfedges and merge both adjacent boundary or vertex cycles), and of their reverse *split_facet* and *split_vertices*. By using the *get_...*, *set_...* and *stitch_...* functions we can write operators that work seamlessly on any kind of HDS (forward, backward, bidirectional, etc.).

As mentioned above, this variability does not come without consequences. For one, the Euler operators must be passed a few more parameters. Most operators take at least

[3] The *compare_...* functions are especially useful to write pre- and post-conditions for the high-level operations.

four arguments, such as *split_facet (h1,h2,g1,g2)* or *join_facets(k1,k2,h1,h2,g1,g2)* with the meaning depicted in Figure 3. For convenience, there are also functions such as *split_facet_after (h1,g1)* if the HDS is forward (the parameters *h2* and *g2* can be deduced), or *join_facets(k1)* if the HDS is bidirectional.

Some preconditions are an integral part of the specifications: for instance, *join_facets* has the precondition that either facet links are not supported, or that the boundary cycles differ for both halfedges. For HDS with holes, this operation is provided and called *split_boundary* and its effect is similar to *join_facets*, except that for HDS with holes, it records the portion containing *g1* as an inner hole of the common adjacent facet. (This part is still under integration in the HDSTL.)

3.3 Memory Layout Reorganization

An important part of designing a data structure is the memory layout. Most implementations impose their layout, but our design goal flexibility implies that we give some control to the user as well. We differentiate with the static design (fixed at compile time, Section 2.7) and the dynamic memory layout reorganization which is the topic of this paragraph.

We say that a HDS is *normalized by facets* if the halfedge belonging to boundary cycles are stored contiguously, in their order along the boundary. This means that the halfedge iterator provided by the HDS will enumerate each facet boundary in turn. In case facets are supported, moreover, *facet normalization* means that the corresponding facets are also stored in the same order. A HDS is *normalized by (source or target) vertices* if a similar condition is satisfied for (source or target) vertex cycles, with similar definition of *vertex normalization*, and *normalized by opposite* if halfedges are stored next to their opposite. These algorithms can only be provided if the HDS is halfedge-relocatable. Note that in general, these normalizations are mutually exclusive. Furthermore, they do not get rid of the storage for the pointers (except in the case where the length of the cycle is fixed, such as for opposite pointers, or in triangulations, in which case some implicit scheme can save storage, as explained in Section 2.7). Normalization algorithms can be implemented in linear time.

4 Experimental Evaluation

4.1 Comparison of Various Models

Here we compare various models of HDS provided in the HDSTL. The models can be *compact* (normalized by opposite, hence no storage for opposite pointer), *indices* (handles are integers instead of pointers; dereferencing is more costly, but copying does not necessitate pointer translation), or pointer-based by default. The functionality can be *minimal* (no vertices or facets), *graphic_fw* (vertices, no facets, and forward-only), *graphic* (vertices but no facets), *default* (vertices, facets, but forward-only), and *maximal* (vertices, facets, bidirectional). Moreover, we have the possibility of storing the halfedge incidences at source (*next_at_source*) or at target (*next_at_target*) instead of in facet (*next_in_facet*), as well as choosing between storing the source or target vertex.

We do this in order to measure the differences in run time. We measure no difference with storing incidences at source or at target.

The last three lines illustrate as well other HDS for which we could write adaptors. We have not mentioned runtime ratios, because it is not clear how to provide a fair comparison. The LEDA graph [14] stores vertices, edges, and halfedges in a doubly linked list, and stores four edge pointers per node and per edge, as well as degree per node and other extra information. It is therefore a very powerful, but also very fat structure, and it offers no flexibility. The CGAL models of HDS [7] however, are comparable in storage and functionality, although less flexible. They require for instance facet-based pointer storage (*next* and *prev_in_facet*). We can reasonably estimate that their performance would match the corresponding models in the HDSTL.

The results are shown in Table 1. We measure the running time ratios by effecting a million pairs *split_facet*/*join_facets* in a small (tetrahedron) HDS. Measurements with million pairs *split_vertices*/*join_vertices* are slightly slower, but in similar ratio. There is therefore no effect due to the (cache and main) memory hierarchy. The running times can vary between a relative 0.6 (facet-based minimal using vector storage) to 2.2 (source-based maximal using list storage). The index-based structure is also slower, but not by much, and it has the double advantage that copying can be done directly without handle translation, and that text-mode debugging is facilitated because handles are more readable.

In conclusion, the model of HDS can affect the performance of a routine by a factor of more than 4. In particular, providing less functionality is faster because fewer pointers have to be set up. Of course, if those pointers are used heavily in the later course of a program, it might be better to store them than recompute them. We also notice that the runtime cost of more functionality is not prohibitive, but care has to be taken that the storage does not exceed the main storage available, otherwise disk swapping can occur. When dealing with large data, it is best to hand pick the least memory-hungry model of HDS that fits the requirements of the program. Also, for high-performance computing applications, its might be a good strategy to “downgrade” the HDS (allow bidirectional storage but declare the HDS as forward-only) and in a later pass set the extra pointers.

4.2 Delaunay Triangulation

We programmed the divide-and-conquer Delaunay triangulation [16, 5], with Dwyer's trick for efficiency (first partition in vertical slabs of $\sqrt{n \log n}$ points each, recursively process these slabs by splitting with horizontal lines, then merge these slabs two by two). There does not seem to be a way to do with only forward HDS: when merging two triangulations, we need a clockwise traversal for one and a counter-clockwise for the other; moreover, simply storing the convex hull in a bidirectional list will not do, as we may need to remove those edges and access the edges inside during the merging process. We need vertices (to contain a point) and bidirectional access to the HDS. This algorithm only uses orientation and in-circle geometric tests. We used a custom very small geometry kernel with simple types (array of two floating point *double* for points).

For this reason, we only tried the divide-and-conquer algorithm with three variants of HDS: the graphic HDS (vertices and edges, bidirectional, with vertex links) the compact graphic HDS (graphic without vertex links, with opposite halfedges stored con-

HDS	features	pointers	runtime ratio
default	V+L,FWif,F+L	v+4h+f	0.95
	id., FWas	v+4h+f	1.0
	id., list storage	3v+6h+3f	2.0
indices	default+indices	v+4h+f	1.25
	id., FWas	v+h+f	1.25
compact_fw	FW,compact	h	0.6
compact	BI,compact	2h	0.65
minimal	FWif	2h	0.6
	id. but FWas	2h	0.65
graphic_fw	V+L,FWif	v+3h	0.7
	id., FWas	v+3h	0.75
graphic	V+L,BIif	v+4h	0.7
	id., BIas	v+4h	0.75
maximal	V+L, BIif, F+L	2v+5h+2f	1
	id., BIas	2v+5h+2f	1.1
	id., list storage	4v+7h+4f	2.15
LEDA_graph	V+L, BIas	$\geq$4(v+h)	N/A
CGAL_minimal	FWif	2h	N/A
CGAL_maximal	V+L, BIif, F+L	2v+5h+2f	N/A

Table 1. A synoptic view of the different models of HDS : (a) name, (b) features [V=vertex,F=facet,L=link,FW=forward,BI=bidirectional, if=in_facet,as=at_source], (c) number of pointers required by the structure, (d) runtime ratios for pair split_facet/join_facets.

tiguously), and the maximal HDS (graphic with facets and facet links as well). Used by the algorithm are the vertex information, opposite and bidirectional pointers in facets and at vertices. The algorithm does not use but otherwise correctly sets all the other pointers of the HDS.

The results of our experiments are shown in Table 2. We are not primarily interested in the speed of our implementation, although we note that on a Pentium III 500Mhz with sufficient memory (256MB), triangulating a million points takes about 11 seconds for the base time, which is very competitive. More interestingly, however, we compare again the relative speed of the algorithms with our different models of HDS, and we record both the running time and the memory requirements. The results in Table 2 suggest that the model of the HDS has an incidence on the behavior of the algorithm, although more in the memory requirement than in the running time (there is at most 15% of variation in the running time). Notice how the compact data structure uses less memory and also provokes fewer page faults (measured by the Unix *time* command). Also interesting is the difference between the compact and non-compact versions of the graphic HDS with and without Dwyer's strategy (10% vs. only 2%). These experiments seem to encourage the use of compact data structures (when appropriate).

These experiments also show how the functional requirements of an algorithm limit the choices of HDS that can be used in them. We could not test the graphic-forward

HDS	time	ratio	memory	page faults
graphic_cp	6.635	1.11	22.3MB	24464
graphic	5.95	1	27.9MB	29928
maximal	6.9	1.15	35.6MB	36180
graphic_cp	3.22	1.03	22.3MB	24464
graphic	3.125	1	27.9MB	29928
maximal	3.57	1.145	35.6MB	36180

Table 2. A comparison of different HDS used with divide-and-conquer Delaunay triangulation without (first three lines) or with (last three lines) Dwyer's trick (and 2.10^5 points): (a) running times and (b) ratios, (c) memory requirements, (d) paging behavior.

HDS with this algorithm. When provided, however, the comparison is fundamentally fair since it is the *same* algorithm which is used with all the different structures .

5 Conclusion

We have provided a framework for expressing several (indeed, many) variants of the halfedge data structure (HDS), including geometric graph, winged-edge, Facet/Edge and dual Vertex/Edge structures. Our approach is to specify a set of requirements (refining each other) in order to express the functional power of a certain HDS. This approach is a hallmark of generic programming, already used in the C++ STL, but also notably in the Boost Graph Library and in CGAL[12]. Regarding the latter reference, we have enriched the palette of models that can be expressed in the framework. For instance, we can express holes in facets or non-manifold vertices. Also we do not require any geometry, thus allowing the possibility to use the HDSTL in representing purely combinatorial information (for instance, in representing hierarchical maps in GIS).

As a proof of concept, we offer an implementation in the form of a C++ template library, the HDSTL (halfedge data structure template library), which consists of only 4,500 lines of (admittedly dense) code, but can express a wide range of structures summarized on Table 1. We provide a set of generic algorithms (including elementary manipulations, Euler operators, and memory layout reorganization) to support those structures.

Regarding our goals, flexibility is more than achieved, trading storage cost for functionality: We have provided a rich variety of models in our framework, each which its own strength and functionality. We also have shown that it is possible to provide a set of generic algorithms which can operate on every single one of them. This flexibility is important when the HDS is expressed as a view from within another structure (e.g. solid boundary, lower-dimensional section or projection, dual view) for which we may not control the representation. It is of course always possible to set up a HDS by duplicating the storage, but problems arise from maintaining consistency. Instead, our flexibility allows to access a view in a coherent manner by reusing the HDSTL algorithms, but on the original storage viewed differently.

Ease of use comes from the high-quality documentation provided with the HDSTL, but there are issues with error messages when trying to use unsupported features, or when debugging. This can be improved using recent work in concept checking [17].

Efficiency is acquired by using C++ templates (static binding, which allows further optimizations) instead of virtual functions (dynamic binding, as provided in Java). The algorithms we provide are reasonably efficient (roughly fifteen seconds for Delaunay triangulation of a million points on a Pentium 500Mhz with sufficient main memory).

Acknowledgments. Thanks to Lutz Kettner for many discussions about his design, to Jack Snoeyink for his helping hand in the early debugging process, and to the people in the CGAL project, notably Sylvain Pion and Monique Teillaud, for past discussions and inspiration.

References

1. M. H. Austern. *Generic Programming and the STL*. Professional computing series. Addison-Wesley, 1999.
2. B. G. Baumgart. A polyhedron representation for computer vision. In *Proc. AFIPS Natl. Comput. Conf.*, volume 44, pages 589–596. AFIPS Press, Arlington, Va., 1975.
3. J.-D. Boissonnat, O. Devillers, M. Teillaud, and M. Yvinec. Triangulations in CGAL. In *Proc. 16th Annu. ACM Sympos. Comput. Geom.*, pages 11–18, 2000.
4. E. Brisson. Representing geometric structures in d dimensions: Topology and order. *Discrete Comput. Geom.*, 9:387–426, 1993.
5. C. Burnikel. Delaunay graphs by divide and conquer. Research Report MPI-I-98-1-027, Max-Planck-Institut für Informatik, Saarbrücken, 1998. 24 pages.
6. D. Cazier and J.-F. Dufourd. Rewriting-based derivation of efficient algorithms to build planar subdivisions. In Werner Purgathofer, editor, *12th Spring Conference on Computer Graphics*, pages 45–54, 1996.
7. *The CGAL Reference Manual*, 1999. Release 2.0.
8. J. O. Coplien. *Multi-Paradigm Design for C++*. Addison-Wesley, 1999.
9. M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, Berlin, 1997.
10. E. Flato, D. Halperin, I. Hanniel, and O. Nechushtan. The design and implementation of planar maps in CGAL. In *Abstracts 15th European Workshop Comput. Geom.*, pages 169–172. INRIA Sophia-Antipolis, 1999.
11. L. J. Guibas and J. Stolfi. Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams. *ACM Trans. Graph.*, 4(2):74–123, April 1985.
12. L. Kettner. Using generic programming for designing a data structure for polyhedral surfaces. *Comput. Geom. Theory Appl.*, 13:65–90, 1999.
13. L.-Q. Lee, J. G. Siek, and A. Lumsdaine. The generic graph component library. In *Proceedings OOPSLA'99*, 1999.
14. K. Mehlhorn and S. Näher. *LEDA: A Platform for Combinatorial and Geometric Computing*. Cambridge University Press, Cambridge, UK, 1999.
15. D. E. Muller and F. P. Preparata. Finding the intersection of two convex polyhedra. *Theoret. Comput. Sci.*, 7:217–236, 1978.
16. F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, 3rd edition, October 1990.
17. J. Siek and A. Lumsdaine. Concept checking: Binding parametric polymorphism in C++. In *First Workshop on C++ Template Programming, Erfurt, Germany*, October 10 2000.
18. K. Weiler. *Topological Structures for Geometric Modeling*. PhD thesis, Rensselaer Polytechnic Institute, Troy, NY, August 1986.