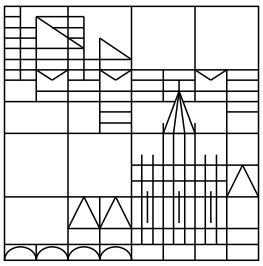

Universität Konstanz

Iterators and Handles for Nodes and Edges in Graphs

Dietmar Kühl
Karsten Weihe

Konstanzer Schriften in Mathematik und Informatik
Nr. 15, September 1996

ISSN 1430–3558

Iterators and Handles for Nodes and Edges in Graphs

Dietmar Kühl[1] Karsten Weihe[1]

Lehrstuhl für praktische Informatik I
(Algorithmen und Datenstrukturen)

15/1996
Universität Konstanz
Fakultät für Mathematik und Informatik

Abstract

In the CGAL startup meeting in Zürich, September 1996, it has turned out that iterators on highly structured container types is a key design issue. This document is intended to serve as an additional basis for further discussions about this topic.

We focus on graph data structures, and we propose a taxonomy of iterators and related types in C++. Moreover, we introduce templates that make defining such classes for a given graph data structure easy.

The concept presented here might offer high flexibility and efficiency. However, for a real library, a good compromise must be found between these desirable goals and ease of use (and ease of learning!). Therefore, our point of view is by no means "puristic." For example, in [4] we developed a proposal for an integration into the LEDA library, which tries to find a good compromise between LEDA's ease–of–use style and our own concept.

Anyway, we think that our concept is a good point to start from, not only in graph algorithmics, but also in computational geometry.

1 Introduction

It is often argued that the Standard Template Library (STL) is not only a library, but also a general design principle [3]. One of the major design principles is a taxonomy of "categories of iterators." "Categories" does not necessarily mean different classes. Rather, it means different sets of *requirements*. The general idea is to define an interface to the underlying container class, which consists of one or more light–weight objects. Algorithms do not work on the plain data structure, but on this interface. In this spirit, we propose a taxonomy of iterators and related types tailored to graph data structures and to the specific needs of graph algorithms. We refer the reader to [1] for an introduction to this concept in view of graph algorithms and for a discussion of its consequences.

Our taxonomy is tuned to be efficient. Whenever efficiency is not a primary concern, these classes may collapse into three classes: HeavyLinNodeIt, HeavyLinEdgeIt, and HeavyAdjIt

[1]Universität Konstanz, Fakultät für Mathematik und Informatik, Postfach 5560/D188, 78434 Konstanz, Germany, {dietmar.kuehl,karsten.weihe}@uni–konstanz.de,
http://www.informatik.uni–konstanz.de/~{kuehl,weihe}

(cf. Sect. 2). The other classes are mainly introduced to save space and to avoid copying heavy–weight classes, as far as this is possible at all. (To our surprise, our practical experiences tell us that this innocent–looking operation may increase the run time of the entire algorithm significantly: by a factor of 2 or more!)

In this manuscript, we assume that the reader is familiar with C++ and with the basic concepts of STL. Moreover, we will sometimes mention *data accessors*. For an introduction to this concept, we refer the reader to [2].

This document is written for developers, not for users, of graph libraries. It might also be interesting for developers of other complex data structures. The document is only preliminary: In the near future, we will compile a more comprehensive manuscript, which also covers Refs. [1, 2, 4].

2 Overview

The taxonomy consists of the categories depicted in Fig. 1.

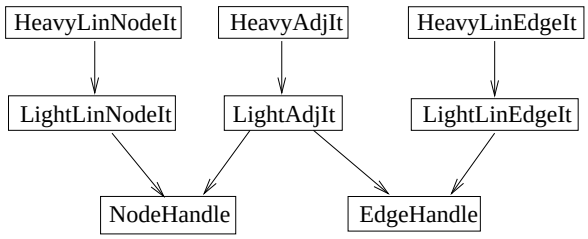

Figure 1: The different categories of iterators and handles. An arrow indicates that the tail fulfills all requirements defined for the head. Note that this diagram does not constitute an inheritance hierarchy (although it could be implemented this way).

First we roughly explain the difference between the different layers:

Handle: A node/edge handle is not assumed to be anything but a mere handle for a single, fixed node or edge. It can be used to mark a node/edge, and a data accessor may use such a handle to access the node's or edge's data.

Light: Objects of these categories are iterators in its true sense. Each category iterates over a certain sequence of nodes or edges. An object of such a category is not assumed to have access to anything but the current item of its sequence and to a reference to the next item. In terms of STL, such an iterator is not assumed to be anything but a forward iterator (although the syntactic requirements are quite different, see Sect. 4).

Heavy: Objects of these categories are assumed to also have some global knowledge about the underlying graph as a whole. For example, this global knowledge allows conversions between objects of different categories (methods `init_heavy` in Sect. 4).

Next we roughly explain the differences between the categories on the same level.

LinNodeIt: Iterates over the nodes of a graph in a linear fashion.

LinEdgeIt: Iterates over the edges of a graph in a linear fashion.

AdjIt: Marks a fixed node and iterates over the edges leaving this node in a linear fashion.

3 Abstract Model

The idea is to implement algorithms on a purely abstract view of graphs. This means that there need not be a one–to–one correspondence between the abstract graph observed by an algorithm and the concrete data structure behind iterators and data accessors. (There need not be a concrete data structure at all!) This abstract graph is defined by handles, iterators, data accessors, and possibly further light–weight classes. This has strong consequences.

- Almost everything that can be interpreted as a graph from an abstract point of view may be used as a graph data structure. For instance, often the set of all edges is too large to be stored explicitly. Instead, edges (and their parameters) must be computed "on the fly." For example, the nodes may be points in the plane, all pairs of nodes are edges, and as a parameter, the length of an edge is the Euclidean distance of the two endpoints. To give a concrete example: This scenario occurs in the Euclidean traveling salesperson problem. Another concrete example was implemented by us as a design case study: The nodes are strings, and two strings are connected by an edge if and only if their Hamming distance equals 1. (This case study is described in detail in the first author's master's thesis.)

- Algorithms may be restricted to subgraphs of the concrete graph data structure. Typically, this means that iterators do not pass all nodes or edges of the graph, only a subset. However, this makes no difference for the algorithm; only a different light–weight iterator class must be defined (or simply instantiated, see Sect. 7 and Ref. [4]).

- Some kind of graph may be interpreted as another kind of graph. For example, many algorithms for directed graphs apply to undirected graphs as well: Either a dummy orientation must be chosen for each edge, or an undirected edge is to be interpreted as a pair of antiparallel directed edges.

- Algorithms may be specialized by special iterator and handle types. For instance, in all definitions in Sect. 4, the order in which an iterator passes all sequence items is unspecified. Special iterator classes (possibly tailored to special classes of graphs) may actually specifiy an order. A good example are adjacency iterators on planar graphs: The order of iteration might reflect the combinatorial embedding; and when stepping over from the tail to the head of an edge (basically a call to method curr_adj), the start item in the adjacency sequence of the head might be the next neighbor on the right or left.

  The documentation of the algorithm may impose such semantic requirements, but of course, there is no way to ensure such a condition (possibly not even a reasonable way to check it at run time).

In summary, it is important to keep in mind that the abstract model on which the following definitions are based is not really a restriction. For instance, the rough definitions of linear node iterator, linear edge iterator, and adjacency iterator at the end of Sect. 2 are statements about the abstract model observed by the algorithm, not about the underlying graph data structure: The abstract graph is just *defined* by the set of nodes and edges seen by these iterators; the set of edges accessible from a node is *defined* by the adjacency iterator (for example, the iterator could pass exactly the edges that *enter* this node in the underlying graph data structure).

4 Detailed Requirements

Every graph algorithm may assume that the individual categories in this taxonomy fulfill the following requirements. (Of course, the concrete signatures of methods are only exemplary.)

NodeHandle: A node handle object provides a method

```
bool valid() const;
```

If `valid()==true`, the object is associated with some fixed node of some fixed graph (whatever this means). Node handle objects are fully compatible with all node data accessors. That is, when implementing an algorithm, one may safely assume that the `get` (and `set` method, if provided) of a node data accessor can be called with a node handle without compile time error. For a valid node handle, these methods access the data of the node associated with this handle.

The default constructor is provided and constructs a non–valid object; copy constructor, `operator=`, `operator==`, and `operator!=` are provided, too. These operators have reference semantics, that is, `operator=` copies only the handle, not the node itself, and `operator==` and `operator!=` check the object identity of the node.

EdgeHandle: completely analogous to NodeHandle.

LightLinNodeIt: A light linear node iterator fulfills all requirements of node handles. In addition, the following methods are provided:

```
bool eol();

LightLinNodeIt const& operator++();
```

The method `eol` (end of list) returns `!valid()`, and the `operator++` returns `*this`. If `!eol()`, `operator++` goes on to the very next item of the sequence, otherwise the behavior is undefined.

In addition, a light linear node iterator provides an implicit conversion to NodeHandle (obvious semantics).

LightLinEdgeIt: completely analogous to LightLinNodeIt.

LightAdjIt: A light adjacency iterator fulfills all requirements of NodeHandle and LinEdgeIt. In addition, the following methods are provided:

```
bool has_node() const;

NodeHandle curr_adj () const;
```

If `has_node()==false`, we also have `valid()==false`. If `has_node()==true`, the light adjacency iterator object is associated with a fixed node. If so and if `valid()==true` in addition, it is also associated with a single edge that leaves this node. Using `operator++`, all edges leaving this node are passed in a linear fashion.

If `valid()==true`, the method `curr_adj` returns an object that points to the target of the current edge, otherwise the behavior is undefined.

In addition, a light adjacency iterator provides implicit conversions to NodeHandle and EdgeHandle (obvious semantics).

HeavyLinNodeIt: A heavy linear node iterator fulfills all requirements of LightLinNodeIt. In addition, the following methods are provided:

```
void init();

void init (NodeHandle const&);

void init_heavy (HeavyLinEdgeIt const&);

void init_heavy (HeavyAdjIt const&);
```

The first method positions the object at the beginning of the linear sequence of all nodes, the second method, on the node referred to by the parameter. These two methods assume that the parameter handles a node/edge of the graph that is "known" by `*this`.

The last two methods do not rely on this precondition. They also overwrite the internal knowledge about the underlying graph. The third method positions `*this` at the beginning of the sequence, and the fourth method, on the parameter.

Moreover, implicit conversions to LightNodeIt and NodeHandle are provided (obvious semantics).

Note: The implicit conversion to NodeHandle must be implemented, too, because user–defined conversions are not transitive in C++.

HeavyLinEdgeIt: completely analogous to HeavyLinNodeIt (with method `init_heavy` overloaded for `HeavyLinNodeIt` and `HeavyAdjIt`). In addition, the following methods are provided (obvious semantics).

```
NodeHandle source() const;

NodeHandle target() const;
```

HeavyAdjIt: A heavy adjacency iterator fulfills all requirements of LightAdjIt and offers method `init_heavy` for `LinNodeIt` and `LinEdgeIt`. In addition, the following methods are provided:

```
void update (NodeHandle const&);

void update (EdgeHandle const&);
```

The former method positions the adjacency iterator on the node referred to by the parameter, and the initial edge is the first item in the sequence of all edges leaving this node. The latter method positions the adjacency iterator on the source of the parameter, and the parameter is also the initial edge. Both methods assume that the parameter handles an object of the same graph.

To illustrate this taxonomy, we consider an example: Dijkstra's algorithm for shortest paths from one root to all nodes. Typically, the input consists of a graph, a node (the root), and the edge lengths. The output consists of distance labels of all nodes.

In our concept, the input consists of a heavy adjacency iterator and two data accessors: The iterator handles the root node, and the data accessors provide access to edge lengths and node distances, respectively. To initialize the node distances, a heavy linear node iterator is constructed from this adjacency iterator. The node priority queue contains node handles. The core loop is an interplay of the categories NodeHandle and HeavyAdjIt, which is "moderated" by the heavy root object. More specifically, whenever a node handle is removed from the queue, it is used to update a HeavyAdjIt that is global to the algorithm and constructed from the root (possibly the root itself). On the other hand, whenever an adjacency iterator is going to be stored in the queue, it is silently converted into a node handle.

Further details of this example are given in [4].

5 Rationale

We assume (and our practical experiences so far justify this assumption) that this large taxonomy allows implementations of graph algorithms which are much more efficient than implementations based on a smaller taxonomy. The distinction between linear node iterators, linear edge iterators, and adjacency iterators is obviously desirable. However, several further questions arise.

1. *Why several layers?*

   Node and edge handles are necessary to manage containers of nodes and edges efficiently. For example, consider the nodes in a priority queue in Dijkstra's algorithm. Managing a queue of node handles not only saves space, but the necessary copy/assignment operations are also much faster.

   The distinction between light and heavy classes is necessary for the same reasons. There are (rare) situations where copy operations are heavily applied to iterators. If the additional features of heavy iterators are not necessary, light iterators may be used to achieve a better performance.

   In our opinion, light iterators are not that important and could be dropped potentially. However, for reasons of efficiency, the distinction between the first and the third layer is a must.

2. *Why no explicit graph object?*

   It is not natural to have heavy iterators, which provide methods for mutual conversions; in fact, it is more natural to have some kind of graph representation that provides methods to construct iterators of all kinds. This potentially removes the need for heavy iterators. However, there are severe problems.

   First of all, in some graph data structures a method like source for HeavyLinEdgeIt might require a strong coupling and heavy collaboration of the linear edge iterator object and the graph data structure. If there is a separate graph object and the linear edge iterator is light, there are chances that this strong coupling cannot be reasonably hidden from the algorithm, which were a disaster. However, this is not a problem at all if the knowledge about the underlying graph data structure is integrated in the (heavy) iterator.

Next consider the situation that an algorithm works on several graph objects simultaneously and maintains a great many of iterators on these different graphs. If the graph objects are separated from the iterators, the algorithm must maintain the correct assignment of iterators to graphs. It goes without saying that this task is highly prone to error, all the more so since there might not even be a reasonable possibility to check correct assignment of light iterators to graphs at run time.

Even more, it is simply not true that such a graph object makes heavy iterators superfluous. For example, an observer iterator [4] that, say, counts the number of calls to `operator++` makes the iterator heavy.

After all, if it turns out that the design of an algorithm can be improved by letting an additional graph object do the job, an iterator may always be used as such an object.

3. *Why not STL style?*

   Whether or not post–increment, `operator++(int)`, is required might be a matter of taste.

   Our iterator classes are not assumed to provide `operator*` for data access. This design decision is extensively discussed in [2]. Moreover, it is not clear what `operator*` should return for adjacency iterators: the node data or the edge data? For the same reason, `operator->` is not assumed.

   A very controversial point in the design of STL was how the end of a sequence is marked. The two alternatives were: (i) an iterator provides a self–check for being "positioned" past the end of the sequence, or (ii) an additional, invalid iterator object indicates the "position" past the end. As is well known, the second alternative was adopted. However, in Sect. 4, we adopt the first alternative. We think this is the better choice here, because algorithms do not—and should not—have direct access to sequences inside the graph data structure, so they cannot call an `end` method to construct such a past–the–end marker.

   Anyway, for heavy iterators it is easy to implement an additional method that constructs a past–the–end marker for the sequence traversed by this iterator object. However, for true light iterators, this might be next to impossible.

4. *Why are graph iterators modeled after STL* **forward** *iterators?*

   For obvious reasons, `operator--` should not be a basic requirement for all potential iterator classes, so graph iterators should not generally fulfill the requirements for bidirectional or random access iterators. On the other hand, it is possible that an algorithm requires only the functionality of an input or output iterator. However, we are not aware of any non–pathological example where such an input or output iterator cannot be easily implemented as a forward iterator.

   Of course, it is no problem to combine the taxonomy of Sect. 2 orthogonally with the taxonomy of STL (input/output/forward/bidirectional/random). Because of this orthogonality, there is no reason for explicitly inserting this possibility in the taxonomy.

6 Templates for Easy Customization

It is certainly tedious and time–consuming to write all the above–mentioned iterator classes from scratch for a graph data structure. But there is no need for that. It is important to observe that most of the code is the same for all common graph data structures. In fact, like for data accessors [2], it is typically possible to factor out all this common stuff into a few class templates. These templates are instantiated with a *traits class*, which provides all specific definitions. For example, assume that your nodes and edges are simply accessed through list pointers (let's call the pointer types NodePtr and EdgePtr). Then the whole taxonomy is immediately available. For instance, one can immediately define:

```
LinNodeIt <ListPointerTraits<NodePtr> >  it;
```

The "magic" behind this definition is rather simple:

```
template <class Ptr>
struct ListPointerTraits
{
  typedef Ptr rep_type;

  static bool is_null  (Ptr ptr)                 { return !ptr; }
  static Ptr  null     ()                        { return 0; }
  static bool is_equal (Ptr ptr1, ptr ptr2)      { return ptr1 == ptr2; }
  static void assign   (Ptr source, Ptr& target) { target = source; }
  static void forward  (Ptr& ptr)                { ptr = ptr->next; }
};
```

The crucial point is: To implement the taxonomy for a graph data structure, it suffices to write traits classes like this for nodes and edges, which, however, must provide the same methods with the same signatures (but, of course, different function bodies). To give an idea how such traits classes implement the whole taxonomy, we give a simplified version of our LightLinNodeIt template as an example. The other handles and iterators are defined analogously.

```
template <class Traits>
class LightLinNodeIt
{
public:

  typedef Traits::rep_type rep_type;

  LightLinNodeIt (rep_type rep=Traits::null()) : _rep(rep) { }

  bool eol   () const { return Traits::is_null (_rep); }
  bool valid () const { return !eol (); }

  LightLinNodeIt const& operator++ () { Traits::forward (_rep); return *this; }

private:

  rep_type  _rep;
};
```

7 Outlook

We also implemented generic decorator classes for iterators and handles [4]. For example, we have implemented *skip iterators*. Roughly speaking, a skip iterator is associated with a predicate defined on the set of all nodes/edges, and `operator++` skips all nodes/edges that do not fulfill this predicate.

All these handle and iterator classes do not provide any means to modify the structure of the graph, for example, to add/remove nodes and edges. For tasks like this, we propose a taxonomy of *mutator categories*, which shall be incorporated in the next release of this document.

References

[1] Kühl, D., and K. Weihe. Using Design Patterns for Reusable Implementations of Graph Algorithms. Konstanzer Schrift Nr. 1 in Mathematik und Informatik.
ftp://www.informatik.uni–konstanz.de/pub/preprints/1996/preprint–001.ps.Z.

[2] Kühl, D., and K. Weihe. Data Access Templates. Konstanzer Schrift Nr. 9 in Mathematik und Informatik.
ftp://www.informatik.uni–konstanz.de/pub/preprints/1996/preprint–009.ps.Z.

[3] Musser, D., and A. Saini. STL Tutorial and Reference Guide. Addison–Wesley, Reading, MA, 1996.

[4] Nissen, M., and K. Weihe. Combining LEDA with Customizable Implementations of Graph Algorithms. Soon available as a Konstanzer Schrift in Mathematik und Informatik (analogous URL).