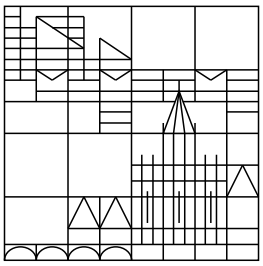


Universität Konstanz

Combining LEDA with Customizable Implementations of Graph Algorithms

Marco Nissen
Karsten Weihe

Konstanzer Schriften in Mathematik und Informatik
Nr. 17, Oktober 1996

ISSN 1430–3558

Combining LEDA with Customizable Implementations of Graph Algorithms

Marco Nissen[1] Karsten Weihe[2]

Lehrstuhl für praktische Informatik I
(Algorithmen und Datenstrukturen)

17/1996
Universität Konstanz
Fakultät für Mathematik und Informatik

Abstract

Efficiency, flexibility, and ease of use are desirable goals in library development, but it seems next to impossible to achieve all three goals simultaneously. The *L*ibrary of *E*fficient *A*lgorithms and *D*ata Structures (*LEDA*) focuses on efficiency and ease of use [4]. On the other hand, in [1, 2, 3], concepts were developed which focus on efficiency and flexibility. In the present manuscript, we introduce a possible integration of these concepts into LEDA.

The main part of this manuscript is a concrete proposal for a possible experimental extension. It is formulated as an additional section of the LEDA user's manual [6], and we used the documentation tool developed for LEDA [5].

The proposal consists of two parts: a toolbox of handle, iterator, and accessor classes [2, 3] and an implementation of the Dijkstra algorithm that realizes the ideas in [1, Sect. 3.1]. Only a restricted version of the toolbox is implemented. In particular, the toolbox does not (yet) support algorithms that change the set of nodes and edges of the underlying graph.

Our concepts are not implemented "puristically." In fact, the proposal is an attempt to find a good compromise between efficiency, flexibility, and ease of use. Moreover, it is a first—necessarily immature—attempt to develop a user–friendly documentation of [1, 2, 3]. It has turned out that the LEDA user's manual and the documentation tool form a good framework for this purpose.

Since LEDA is designed to be easy to use, there is not an urgent need for extensive tutorials for the LEDA features: The reference manual *is* a sufficient tutorial. However, we think that for the concepts in [1, 2, 3] a carefully written tutorial were a great help. Unfortunately, such a tutorial cannot be seamlessly integrated into a reference manual like LEDA's and is hence beyond the scope of this manuscript.

[1]Im Stadtwald, Gebäude 46.1, 66123 Saarbrücken, Germany, marco@mpi–sb.mpg.de, http://www.mpi–sb.mpg.de/~marco. This work was done while the first author was visiting the research group *Algorithms and Data Structures* at the Universität Konstanz.

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

We also evaluated the performance of our implementations. Figs. 1–3 compare the implementation of the Dijkstra algorithm in LEDA (function *DIJKSTRA* [6, Sect. 7.20.2]) with the customization of our implementation to the same organization of all data (function *dijkstra_pred*, Sect. 1.16.2 below).

More specifically, Figs. 1 and 2 are drawn from random directed graphs with 1000 nodes and a varying number of edges. These random graphs are constructed by the following procedure: First a Hamiltonian cycle is inserted, which ensures that the graph is strongly connected; then random directed triangles are inserted until the desired number of edges is (approximately) reached. Such a randomly generated triangle is inserted only if this insertion does not produce parallel edges. Therefore, the resulting graph is simple. Fig. 3 is drawn from random planar triangulations produced by the LEDA function *triangulated_planar_graph* [6, Sect. 7.18]. In all three cases, the edge lengths are uniformly randomly chosen from the interval $[0, 200 \cdot 2^{16}]$.

Figs. 1 and 3 on the one hand, and Fig. 2 on the other hand, show two different scenarios: Figs. 1 and 3 depict the asymptotic complexity of a single execution of the Dijkstra algorithm on large inputs; Fig. 2 depicts the accumulated complexity of a large number of executions of the Dijkstra algorithm on one single graph. To be precise, it shows the run time of an all–pairs shortest path routine, which simply calls Dijkstra once for every node.

In each figure, the solid line (labeled 'LEDA') shows the run time of the LEDA function *DIJKSTRA*. The line labeled 'Handler' shows the run time of the customization of our function *dijkstra_pred* to exactly the same situation, that is, the underlying graph is represented by the LEDA type *graph*, and edge lengths and node distances are organized as LEDA edge/node arrays. This means that we used *handler accessors* to customize *dijkstra_pred*. Finally, the line labeled 'Member' shows the run time of a slightly different scenario: The graph is represented by the parameterized LEDA type *GRAPH*, and edge lengths and node distances are the template parameters. Therefore, here we used *member accessors* to customize function *dijkstra_pred*.

These experiments were done on a Sparc Server 1000 with 384MB RAM, under SunOS. However, we did much more comprehensive experiments, partially on less powerful Sun architectures. The results were very similar. Hence, we think the following general tendency can be safely concluded:

- Current compilers are not (yet?) aggressive enough to optimize all the overhead away.
- However, the overhead might be acceptable for almost all applications,
- the overhead seems to be bounded by a constant factor,
- and seems to have a small variance.

Acknowledgement

We are greatly indebted to Kurt Mehlhorn, Stefan Näher, and Christian Uhrig. This thread of research owes its very existence to their strong engagement. Moreover, we thank them for all the stimulating and fruitful discussions at the MPI and for a lot of constructive criticism.

We also thank Dietmar Kühl for his permanent interest and for his valuable advices.

References

[1] Dietmar Kühl and Karsten 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] Dietmar Kühl and Karsten 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] Dietmar Kühl and Karsten Weihe. Iterators and Handles for Nodes and Edges in Graphs. Konstanzer Schrift Nr. 15 in Mathematik und Informatik.
ftp://www.informatik.uni–konstanz.de/pub/preprints/1996/preprint–015.ps.Z.

[4] Kurt Mehlhorn and Stefan Näher. LEDA, A Platform for Combinatorial and Geometric Computing. Communications of the ACM **38** (1995), 96–102.

[5] Kurt Mehlhorn and Stefan Näher. Manual Pages and Documentation. Chapter 1 of the forthcoming book, *The LEDA Platform of Combinatorial and Geometric Computing.*
http://www.mpi-sb.mpg.de/~mehlhorn/LEDAbook.html.

[6] Kurt Mehlhorn, Stefan Näher, and Christian Uhrig. The LEDA User Manual. Version R 3.4.
http://www.mpi-sb.mpg.de/LEDA/articles.

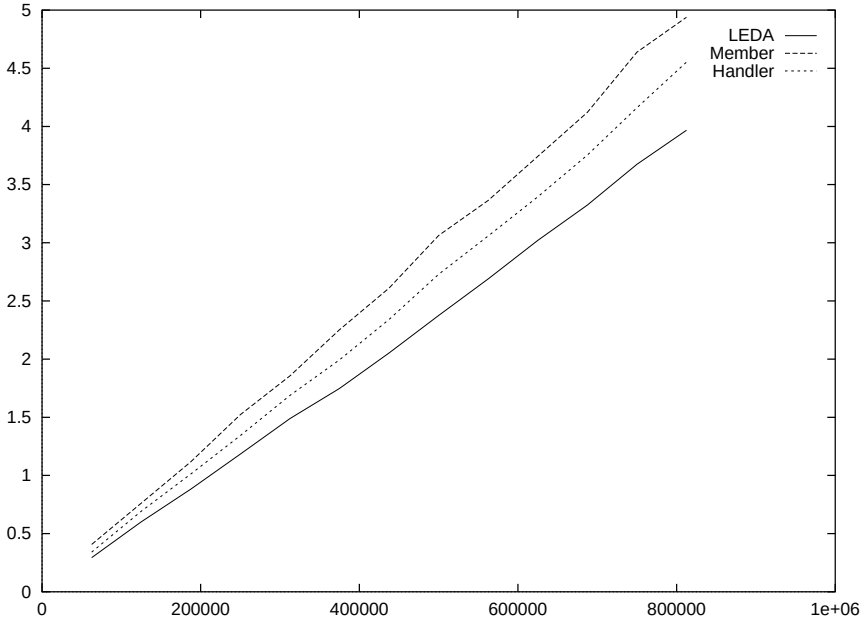


Figure 1: single–source shortest paths on random simple graphs with 1000 nodes (number of edges/run time in seconds).

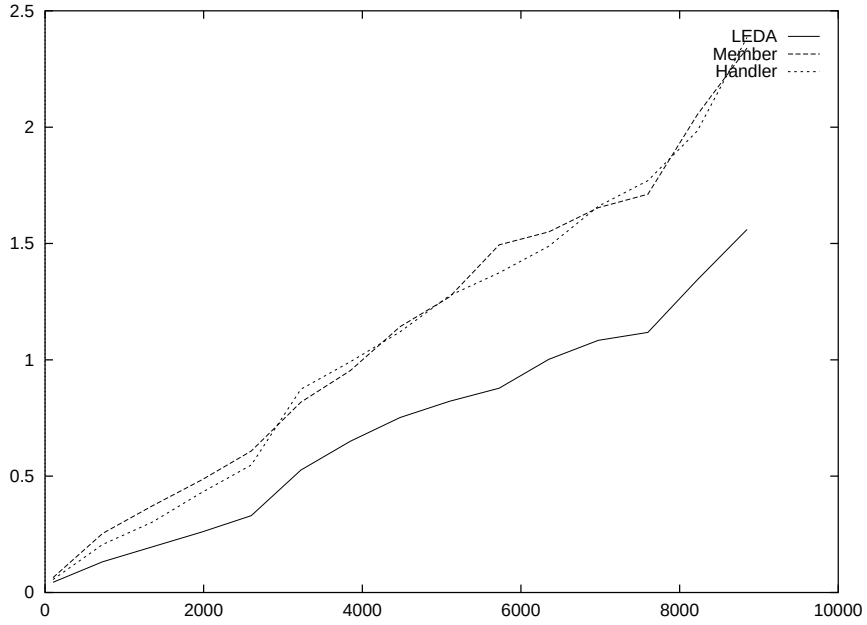


Figure 2: all–pairs shortest paths on random simple graphs with 100 nodes (number of edges/run time in seconds).

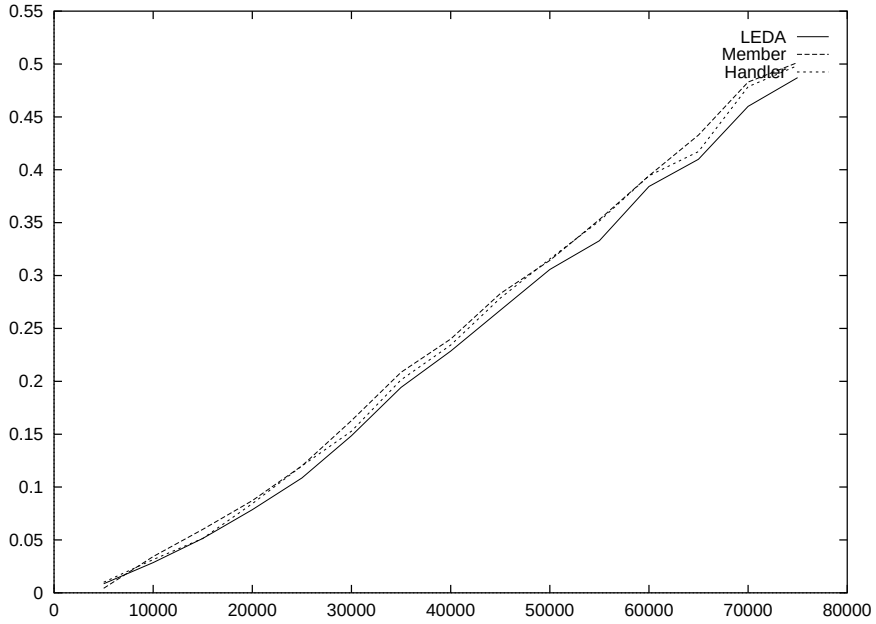


Figure 3: single–source shortest paths on random triangulated planar graphs (number of nodes/run time in seconds).

Contents

1 Customizable Graph Algorithms

Often the implementation of an algorithm does not fit perfectly into the context in which the algorithm is to be applied. This may enforce overhead both in development/maintenance and in run time. In Sects. 1.1–1.14 we introduce new concepts to overcome this problem. Two key problems are addressed: (i) adaptation to specific details of the underlying graph representation and (ii) flexible variations of the behavior of an algorithm. Sect. 1.1 defines a toolbox, which makes adaptation easy. Sect. 1.16 gives an implementation of the Dijkstra algorithm, which is based on this toolbox and whose behavior is kept flexible. The key concept to achieve a high degree of flexibility is that the algorithm is implemented as a class rather than as a subroutine.

1.1 Toolbox

The toolbox consists of a couple of light–weight classes. These classes together form an interface to the underlying graph data structure, and algorithms can be implemented on top of this interface. Each of these light–weight classes is responsible for making a specific aspect of this data structure visible to algorithms.

Basically, there are three types of light–weight classes: handles, iterators and data accessors.

- A node (respectively, edge) handle "marks" a fixed node (edge) and allows restricted access to it (Sect. 1.2 and Sect. 1.3).

- An iterator is a handle whose node/edge is not fixed. Instead, an iterator allows to sequentially traverse all nodes or edges or a subset of nodes or edges. The toolbox provides iterator classes that realize the standard ways of iterating over a graph: in a linear fashion over all nodes or edges and over the nodes/edges adjacent/incident to a fixed node (Sect. 1.4, 1.5, and 1.6, respectively). Moreover, additional iterator classes are provided, which can be "wrapped around" existing iterator classes and change the iterator's functionality (skip iterator, Sect. 1.7 and observer iterator, Sect. 1.8). However, iterator classes themselves do not provide means to access the data associated with a node or edge; this is the data accessors' responsibility.

- An object of a data accessor class is responsible for a specific node or edge parameter. A data accessor provides means, given a handle or an iterator, to read and write its associated parameter for the node/edge to which the handle or iterator currently refers. The toolbox provides data accessor classes for the standard ways to organize node and edge parameters in a graph: in structs attached to the individual nodes and edges (Sect. 1.11, in LEDA: template parameters of GRAPH) or in additional containers outside the graph data structure (Sect. 1.10, in LEDA: node and edge arrays).

  Moreover, the data accessor concept can be used to perform "on–the–fly" calculations of parameter values so that this computation is hidden from all algorithms. Data accessors for two common cases are provided: constant parameter values (Sect. 1.13) and edge parameters computed from node values (Sect. 1.14).

  See Sect. 1.9 for a general overview of data accessors.

All classes in the toolbox are templates. Customizing an algorithm which is based on the toolbox means to instantiate these templates with proper types. For LEDA graphs, all necessary types are predefined. Their definitions are given below and can be used immediately. Moreover, they may serve as an example for defining these types for other graph classes.

Besides that, iterators and data accessors can be used to inspect the algorithm during its execution **without need to modify the code of the algorithm**. For example, this feature can be used for on–line checking, operation counting, and simultaneous graphical animation of an algorithm. The toolbox provides additional generic iterator and data accessor classes for such purposes.

1.2 Node Handle (NodeHandle)

1. Definition

An instance *nh* of class *NodeHandle<traits>* "marks" a fixed node and allows restricted access to it.

2. Creation

NodeHandle<traits> nh(*traits*::*nodetype n* = *traits*::*node_null*( ));
creates an instance *nh* of this class and binds it to *n*.

The template parameter class `traits` must provide several types and static function methods:

- type `nodetype`

  defines the type of the node representation

- `nodetype node_null()`

  returns a characteristic 'nil'-node

- `bool is_node_null(const nodetype& n)`
  returns `true` if and only if n is equal to `node_null()`
- `void assign(nodetype& n1, const nodetype& n2)`
  assigns the second node to the first node.
- `bool is_equal(const nodetype& n1, const nodetype& n2)`
  returns `true` if both nodes are equal.

3. Operations

bool	*nh*.valid( )	returns `true` if *nh* is bound to a node.
nodetype	*nh*.get_node( )	returns the node representation.

4. Implementation

Creation of a handle and all methods take the same time as the underlying operations defined in the traits class.

5. Example

See the example clause for `LinNodeIt` (Sect. 1.4) for a definition of the traits class `LedaNode`.

```
typedef NodeHandle<LedaNode>    nodehandle;
node            n;
nodehandle      nh(n);
```

1.3 Edge Handle (EdgeHandle)

1. Definition

An instance *eh* of class *EdgeHandle*<*traits*> "marks" a fixed edge and allows restricted access to it.

2. Creation

EdgeHandle<*traits*> *eh*(*traits*::*edgetype e* = *traits*::*edge_null*( ));
creates an instance *eh* of this class and binds it to *e*.

The template parameter class `traits` must provide several types and static function methods:

- type `edgetype`
  defines the type of the edge representation
- `edgetype edge_null()`
  returns a characteristic 'nil'-edge
- `bool is_edge_null(const edgetype& e)`
  returns `true` if and only if e is equal to `edge_null()`

- `void assign(edgetype& e1, const edgetype& e2)`

  assigns the second edge to the first edge.

- `bool is_equal(const edgetype& e1, const edgetype& e2)`

  returns `true` if both edges are equal.

3. Operations

bool	*eh*.valid( )	returns `true` if *eh* is bound to an edge.
edgetype	*eh*.get_edge( )	returns the edge representation.

4. Implementation

Creation of a handle and all methods take the same time as the underlying operations defined in the traits class.

5. Example

See the example clause for `LinEdgeIt` (Sect. 1.5) for a definition of the traits class `LedaEdge`.

```
typedef EdgeHandle<LedaEdge>    edgehandle;
edge            e;
edgehandle      eh(e);
```

1.4 Linear Node Iterators (LinNodeIt)

1. Definition

An instance *it* of class *LinNodeIt*<*traits*> is a linear node iterator that iterates over the node set of a graph; the current node of an iterator object is said to be "marked" by this object.

2. Creation

LinNodeIt<*traits*> *it*(*traits*::*graphtype* *G*, *traits*::*nodetype* *n* = *traits*::*node_null*( ));

creates an instance *it* of this class bound to *G* and *n*.
The node and graph are initialized by the parameter list. If the given node is initialized to the null-node, it is assigned the first node in the graph.
Precondition: if not `traits::is_node_null(n)`, *n* is a node of *G*.

The template parameter class `traits` must provide several types and static function methods:

- type `nodetype`

  defines the type of the node representation

- type `graphtype`

  defines the type of the graph representation

- `void forward(const graphtype& G, nodetype& n)`

  performs one step forward in the sequence;

  precondition: n is a node of the graph G

  postcondition: if n was the very last list item, n is set to `node_null()`, otherwise n is now the successor of the former n.

- `nodetype node_null()`

  returns a characteristic 'nil'-node

- `bool is_node_null(const nodetype& n)`

  returns `true` if and only if n is equal to `node_null()`

- `nodetype first_node(const graphtype& G)`

  returns the first node of the given graph G

- `void assign(nodetype& n1, const nodetype& n2)`

  assigns the second node to the first node.

- `bool is_equal(const nodetype& n1, const nodetype& n2)`

  returns `true` if both nodes are equal.

3. Operations

bool	*it*.eol( )	returns `true` if there is no successor node left, i.e. if all nodes of the node set are passed (eol: end of list).
bool	*it*.valid( )	returns `true` iff end of sequence not yet passed, i.e. if there is a node in the node set that was not yet passed.
traits::*nodetype*	*it*.get_node( )	returns the node representation.
traits::*graphtype*	*it*.get_graph( )	returns the graph representation.
LinNodeIt&	++*it*	performs one step forward in the sequence. If there is no successor node, `eol()` will be `true`. This method returns `*this`.

4. Implementation

Creation of an iterator and all methods take the same time as the underlying operations defined in the traits class.

5. Example

The following is an example for a traits class as required by `LinNodeIt`, which yields iterators for the LEDA class `graph`:

```
struct LedaNode {
  typedef node          nodetype;
  typedef graph         graphtype;
  static void forward(const graphtype& G, nodetype& n) {
        n=G.succ_node(n); }
  static nodetype node_null() { return nil; }
```

```
  static bool is_node_null(const nodetype& n) { return n==nil; }
  static const nodetype& first_node(const graphtype& G) {
        return G.first_node(); }
  static void assign(nodetype& n1, const nodetype& n2) { n1=n2; }
  static bool is_equal(const nodetype& n1, const nodetype& n2) { return n1==n2; }
};
```

In defining a traits class for `LinNodeIt`, this syntax must be strictly obeyed. This example has been implemented and is available for immediate use with `LedaLinNodeIt`, which is actually a synonym for `LinNodeIt<LedaNode>`.

1.5 Linear Edge Iterators (LinEdgeIt)

1. Definition

An instance *it* of class *LinEdgeIt*<*traits*> is a linear edge iterator that iterates over the edge set of a graph; the current edge of an iterator object is said to be "marked" by this object.

2. Creation

LinEdgeIt<*traits*> *it*(*traits*::*graphtype* *G*, *traits*::*edgetype* *e* = *traits*::*edge_null*( ));

creates an instance *it* of this class. The edge and graph are initialized by the parameter list. If the given edge is initialized to the null-edge, it is assigned the first edge in the graph.
Precondition: if not `traits::is_edge_null(e)`, e is an edge of G.

The template parameter class `traits` must provide several types and static function methods:

- type `edgetype`

  defines the type of the edge representation

- type `graphtype`

  defines the type of the graph representation

- `void forward(const graphtype& G, edgetype& e)`

  performs one step forward in the sequence;

  precondition: e is an edge of the graph G.

  postcondition: if e was the very last list item, e is set to `edge_null()`, otherwise e is now the successor of the former e.

- `edgetype edge_null()`

  returns a characteristic 'nil'-edge

- `bool is_edge_null(const edgetype& e)`

  returns `true` if and only if e is equal to `edge_null()`

- `edgetype first_edge(const graphtype& G)`

  returns the first edge of the given graph G

- `void assign(edgetype& e1, const edgetype& e2)`

  assigns the second edge to the first edge.

- `bool is_equal(const edgetype& e1, const edgetype& e2)`

  returns `true` if both edges are equal.

3. Operations

bool	*it*.eol( )	returns `true` if there is no successor edge left, i.e. if all edges of the edge set are passed (eol: end of list).
bool	*it*.valid( )	returns `true` iff end of sequence not yet passed, i.e. if there is a edge in the edge set that was not yet passed.
traits::*edgetype*	*it*.get_edge( )	returns the edge representation.
traits::*graphtype*	*it*.get_graph( )	returns the graph representation.
LinEdgeIt&	++*it*	performs one step forward in the sequence. If there is no successor edge, `eol()` will be `true`. This method returns `*this`.

4. Implementation

Creation of an iterator and all methods take the same time as the underlying operations defined in the traits class.

5. Example

The following is an example for a traits class as required by `LinEdgeIt`, which yields iterators for the LEDA class `graph`:

```
struct LedaEdge {
  typedef edge          edgetype;
  typedef graph         graphtype;
  static void forward(const graphtype& G, edgetype& e) {
        e=G.succ_edge(e); }
  static edgetype edge_null() { return nil; }
  static bool is_edge_null(const edgetype& e) { return e==nil; }
  static const edgetype& first_edge(const graphtype& G) {
        return G.first_edge(); }
  static void assign(edgetype& e1, const edgetype& e2) { e1=e2; }
  static bool is_equal(const edgetype& e1, const edgetype& e2) { return e1==e2; }
};
```

In defining a traits class for `LinEdgeIt`, this syntax must be strictly obeyed. This example has been implemented and is available for immediate use with `LedaLinEdgeIt`, which is actually a synonym for `LinEdgeIt<LedaEdge>`.

1.6 Adjacency Iterators (AdjIt)

1. Definition

An instance *it* of class *AdjIt*<*traits*> is an adjacency iterator that marks a node (which is fixed in contrast to linear node iterators) and iterates over the edges that leave this node.

2. Creation

AdjIt<traits> it(traits::graphtype G, traits::nodetype n = traits::node_null( ));

creates an instance *it* of this class. The node and graph are initialized by the parameter list. If the given node is initialized to the null-node, it is assigned the first node in the graph. The edge will be the first incident edge of the node.
Precondition: if not `traits::is_node_null(n)`, n is a node of G.

The template parameter class `traits` must provide several types and static function methods:

- type `nodetype`

  defines the type of the node representation

- type `edgetype`

  defines the type of the edge representation

- type `graphtype`

  defines the type of the graph representation

- `void forward(const graphtype& G, nodetype& n, edgetype& e)`

  performs one steps in the list of incident edges of the current fixed node.

  precondition: n and e belong to G, and n is the source node of e.

  postcondition: if e was the very last list item, e is set to `edge_null()`, otherwise e is now the successor of the former e.

- `void curr_adj(nodetype& n, edgetype& e, const graphtype& G)`

  computes the node and edge that are needed for update of the iterator to be the current adjacent iterator; it is constructed with the target node of the current edge and the first incident edge of it.

  precondition: n and e belong to G, n is the source node of e.

  postcondition: n and e belong to G, n is the source node of e.

- `nodetype node_null()`

  returns a characteristic 'nil'-node

- `edgetype edge_null()`

  returns a characteristic 'nil'-edge

- `bool is_node_null(const nodetype& n)`

  returns `true` if and only if n is equal to `node_null()`

- `bool is_edge_null(const edgetype& e)`

  returns `true` if and only if e is equal to `edge_null()`

- `edgetype first_incident_edge(const graphtype& G, const nodetype& n)`

  returns the first incident edge of a node

  precondition: n is a node of the graph G

  postcondition: the returned edge is an edge of the graph G; the source node of e is equal to n

- `nodetype first_node(const graphtype& G)`

  returns the first node of the given graph G

- `void assign(nodetype& n1, const nodetype& n2)`
  assigns the second node to the first node.
- `void assign(edgetype& e1, const edgetype& e2)`
  assigns the second edge to the first edge.
- `bool is_equal(const nodetype& n1, const nodetype& n2)`
  returns `true` if both nodes are equal.
- `bool is_equal(const edgetype& e1, const edgetype& e2)`
  returns `true` if both edges are equal.

3. Operations

void	*it*.update(*traits*::*nodetype* *n*, *traits*::*edgetype* *e* = *traits*::*edge_null*( ))	updates the current iterator to the given node and edge. If `traits::is_edge_null(e)`, it is initialized with the first edge leaving *n*. **Preconditions:** *n* is a node of the graph to which 'it' is currently bound. If *e* is specified and not `traits::is_edge_null(e)`, then *e* leaves *n*.
bool	*it*.eol( )	returns `true` if there is no successor edge left, i.e. if all edges of the edge set are passed (eol: end of list).
bool	*it*.valid( )	returns `true` iff end of sequence not yet passed.
traits::*edgetype*	*it*.get_edge( )	returns the edge representation.
traits::*nodetype*	*it*.get_node( )	returns the node representation.
traits::*graphtype*	*it*.get_graph( )	returns the graph representation.
AdjIt	*it*.curr_adj( )	returns a new adjacency iterator that is constructed with the target node of the current edge and the first incident edge of it.
AdjIt&	++*it*	performs one steps in the list of incident edges of the current fixed node. This method returns `*this`.

4. Implementation

Creation of an iterator and all methods take the same time as the underlying operations defined in the traits class.

5. Example

The following is an example for a traits class as required by `AdjIt`, which yields iterators for the LEDA class `graph`:

```
struct LedaNodeEdge {
  typedef edge          edgetype;
  typedef node          nodetype;
  typedef graph         graphtype;
```

```
  static nodetype node_null()  { return nil; }
  static edgetype edge_null() { return nil; }
  static bool is_node_null(const nodetype& n) { return n==nil; }
  static bool is_edge_null(const edgetype& e) { return e==nil; }
  static const nodetype& first_node(const graphtype& G) {
            return G.first_node(); }
  static void forward(const graphtype& G, nodetype&, edgetype& e) {
        e=G.adj_succ(e); }
  static edgetype first_incident_edge(const graphtype& G,nodetype n) {
        return G.first_adj_edge(n); }
  static void assign(nodetype& n1, const nodetype& n2) { n1=n2; }
  static void assign(edgetype& e1, const edgetype& e2) { e1=e2; }
  static bool is_equal(const edgetype& e1, const edgetype& e2) { return e1==e2; }
};
```

In defining a traits class for `AdjIt`, this syntax must be strictly obeyed. This example has been implemented and is available for immediate use with `LedaAdjIt`, which is actually a synonym for `AdjIt<LedaNodeEdge>`.

1.7 Skip Iterators (SkipLinNodeIt)

1. Definition

An instance *it* of class *SkipLinNodeIt*<*Predicate*, *Iter*> is a linear node skip iterator that iterates over the node set of a graph; the current node of an iterator object is said to be "marked" by this object. The internal predicate object of type `Predicate` controls the access of the iterators: Nodes that do not fulfill the Boolean condition represented by `Predicate` are skipped by `operator++`.

Note: Copy constructor, `operator=`, and `operator==` of the iterator class apply the corresponding methods of the predicate class. To achieve reference semantics, the predicate class must be implemented as some kind of smart pointer (see Example clause in Sect. 1.8).

There are implementations for all kinds of iterators:

- `SkipLinNodeIt`
- `SkipLinEdgeIt`
- `SkipAdjIt`

Each skip iterator offers the same interface as the corresponding base iterator class – i.e. `LinNodeIt`, `LinEdgeIt`, and `AdjIt`, respectively. The correspondence between base iterator class and skip iterator class is completely analogous in all three cases. In the following, we will describe only `SkipLinNodeIt` in detail.

2. Creation

SkipLinNodeIt<*Predicate*, *Iter*> *it*(*Predicate pred*, *Iter base_it*);

creates an instance *it* of this class, which is bound to predicate *pred* and iterator *it*. The iterator is positioned on the first node that fulfills the predicate *pred* (if there is no such node, `it.eol()` is immediately `true`).

Preconditions: `Predicate` must provide a method `operator()` for iterators of type `Iter` that returns a Boolean value and `Iter` has the same requirements as `LinNodeIt<>` (analogous requirement for `SkipLinEdgeIt<>` and `SkipAdjIt<>`).

3. Operations

bool	*it*.eol( )	returns `true` if there is no successor node left, i.e. if all nodes of the node set are passed.
bool	*it*.valid( )	returns `true` iff end of sequence not yet passed, i.e. if there is a node in the node set that was not yet passed.
Iter::*nodetype*	*it*.get_node( )	returns the node representation.
Iter::*graphtype*	*it*.get_graph( )	returns the graph representation.
SkipLinNodeIt&	++*it*	performs one step forward in the sequence. If there is no successor node that fulfills *pred*, `eol()` will be `true`. Otherwise, the iterator is positioned on the first successor node that fulfills *pred*. This method returns `*this`.

4. Implementation

Constant overhead.

5. Example

The following predicate class simply yields `true` for all iterators, such that every node or edge will be seen by any algorithm:

```
template<class Iter>
struct predicate {
  predicate() {}
  bool operator() (const Iter& it) const {
     return true; }
};
```

More examples for predicates can be seen in Sect. 1.15.

1.8 Observer Iterators (ObsLinNodeIt)

1. Definition

An instance *it* of class *ObsLinNodeIt*<*Obs*, *Iter*> is an observer iterator that executes the methods `notify_constructor()` and `notify_forward()` of the class `Obs` for each iterator operation. These observer iterators can be used e.g. for operation–counting, on–line checking or animation. For this aim, the methods `notify_constructor()` and `notify_forward()` get the current iterator as input parameter. For example, if the nodes are points in the plane and an algorithm animation shall always highlight the current point, the observer object may "ask" the current point's coordinate from iterator to highlight the corresponding point on the screen.

Note: Copy constructor, `operator=`, and `operator==` of the iterator class apply the corresponding methods of the observer class. To achieve reference semantics, the observer class must be implemented as some kind of smart pointer (see Example clause in Sect. 1.8).

Note that the code of the algorithm to be animated is not affected at all by that ! There are implementations for all kinds of iterators:

- `ObsLinNodeIt`
- `ObsLinEdgeIt`
- `ObsAdjIt`

Each observer iterator offers the same interface as the corresponding base iterator class – i.e. `LinNodeIt`, `LinEdgeIt`, and `AdjIt`, respectively. The correspondence between base iterator class and observer iterator class is completely analogous in all three cases. In the following, we will describe only `ObsLinNodeIt` in detail.

2. Creation

ObsLinNodeIt<*Obs*, *Iter*> *it*(*Obs*& *obs*, *Iter base_it*);
creates an instance *it* of this class bound to *obs* and *base_it*. **Preconditions:** `Obs` must provide some methods (see below) and `Iter` fulfills the same requirements as `LinNodeIt<>` (analogous requirement for `ObsLinEdgeIt<>` and `ObsAdjIt<>`).

The template parameter class `Obs`, the observer class, must provide several types and static function methods:

- copy constructor
- `operator=`, `operator==`
- `void notify_constructor(const Iter&)`
  This method is always executed when a constructor of *ObsLinNodeIt*<*Obs*, *Iter*> was called.
- `void notify_forward(const Iter&)`
  This method is always executed when the method `operator++` was called.

3. Operations

bool	*it*.eol( )	returns `true` if there is no successor node left, i.e. if all nodes of the node set are passed.
bool	*it*.valid( )	returns `true` iff end of sequence not yet passed, i.e. if there is a node in the node set that was not yet passed.
Iter :: *nodetype*	*it*.get_node( )	returns the node representation.
Iter :: *graphtype*	*it*.get_graph( )	returns the graph representation.
ObsLinNodeIt&	++*it*	performs one step forward in the sequence. If there is no successor node, `eol()` will be `true`. This method returns `*this`.
Obs	*it*.get_observer( )	gives access to the internal observer object.

4. Implementation

Constant overhead.

5. Example

First two simple observer classes. The first one is a dummy class, which ignores all notifications. The second one merely counts the number of calls to `operator++` for all iterators that share the same observer object through copy construction or assignment (of course, a real implementation should apply some kind of reference counting or other garbage collection).

```
template <class Iter>
  class DummyObserver {
  int* _count;
public:
  DummyObserver() : _count(new int(0)) { }
  void notify_constructor(const Iter& ) { }
  void notify_forward(const Iter& ) { }
  int  counter() const { return *_count; }
  int* counter_ptr() const { return _count; }
};

template <class Iter, class Observer>
class SimpleCountObserver {
  int*  _count;
public:
  SimpleCountObserver() : _count(new int(0)) { }
  SimpleCountObserver(Observer& obs) :
        _count(obs.counter_ptr()) { }
  void notify_constructor(const Iter& ) { }
  void notify_forward(const Iter& ) { ++(*_count); }
  int  counter() const {  return *_count; }
  int* counter_ptr() const { return _count; }
};
```

Next an exemplary application, which counts the number of calls to `operator++` of all adjacency iterator objects inside `dummy_algorithm`. Here the dummy observer class is used only as a "Trojan horse," which carries the pointer to the counter without affecting the code of the algorithm. See Sect.1.4 and 1.6 for definitions of `LedaNode` and `LedaNodeEdge`.

```
template<class Iter>
bool break_condition (const Iter&) { return false; }

template<class LinNodeIter,class AdjIter>
void  dummy_algorithm(LinNodeIter& it,AdjIter& it2) {
  while (it.valid()) {
    for (it2.update(it); it2.valid() && !break_condition(it2); ++it2) ;
      ++it;
  }
}

int write_count (const graph& G) {
```

```
  typedef DummyObserver<LedaLinNodeIt>                DummyObs;
  typedef SimpleCountObserver<LedaAdjIt,DummyObs>     CountObs;
  typedef ObsLinNodeIt<DummyObs,LedaLinNodeIt>        LinNodeIter;
  typedef ObsAdjIt<CountObs,LedaAdjIt>                AdjIter;

  DummyObs observer;
  LinNodeIter   it(observer,LedaLinNodeIt(G));
  CountObs observer2(observer);
  AdjIter       it2(observer2,LedaAdjIt(G));
  dummy_algorithm(it,it2);
  return it2.get_observer().counter();
}
```

1.9 Data Accessors

An object of a data accessor class is responsible for a single node or edge parameter. This parameter may be stored in different ways: as part of the node (edge) data structure, in an external array, implicitly, or whatsoever. Different data accessor types are able to cover these different scenarios. Given a handler or iterator of nodes (edges) and a data accessor for a node (edge) parameter, an algorithm can read and write this parameter for the node (edge) marked by the handler or iterator. For this aim, functions `get()` for reading and `set()` for writing are provided for every data accessor type (and must be provided for user–defined data accessor types, too).

The signatures of these functions are standardized, which makes it easy to change the organization of the parameter in the underlying data structure **without need to change the code of the algorithms**. Just instantiate the functions with another data accessor type as template parameter.

More specifically, for each data accessor class DA, the following function template is defined:

```
T get (DA da, Iter it);
```

This method returns the value associated with it for the parameter associated with da.

In addition, for all data accessor classes for which this makes sense, the following function template is defined, too:

```
void set (DA da, Iter it, T value);
```

The effect is that `get(da,it)==value` afterwards.

In both cases, `T` and `Iter` are template parameters of `get()` and `set()`. If `DA` manages a node parameter, a type `Iter::nodetype` and a method

```
Iter::nodetype Iter::get_node () const
```

are required for `Iter`, otherwise a type `Iter::edgetype` and a method

```
Iter::edgetype Iter::get_edge () const
```

The handle and iterator classes defined in Sects. 1.2–1.6 fulfill these requirements. Therefore, they can be used immediately with all data accessor types introduced in Sects. 1.10–1.14.

1.10 Handler Accessors (HandlerAccessor)

1. Definition

An instance *DA* of class *HandlerAccessor*<*T*, *Handler*, *StructAccessor*> is instantiated with a type `Handler` that realizes a mapping from nodes or edges to `T`, e.g. `node_array<T>` or `edge_array<T>`.

The data which is associated to the iterators can be accessed by the functions `get()` and `set()` that take as parameters an instance of *HandlerAccessor*<*T*, *Handler*, *StructAccessor*> and an iterator, see below.

2. Creation

HandlerAccessor<*T*, *Handler*, *StructAccessor*> *DA*(*Handler handler*, *StructAccessor sa*);
creates an instance *DA* of this class bound to *handler* and *sa*.

Precondition: for class `StructAccessor` there is a special implementation of the function `get_object()` for every handler and iterator class that is potentially used with *DA*. For example, if `get_object()` is realized as a template of handle/iterator, the requirement is as follows:

```
template<class Iterator>
objecttype get_object(StructAccessor, const Iterator& it);
```

The following definitions of `LedaNodeAccessor` and `LedaEdgeAccessor` fulfill the requirements stated above. They have been implemented and are available for immediate use with all iterators that provide a method `get_node` or `get_edge` like the methods provided by `NodeHandle<>` and `EdgeHandle<>`.

```
struct LedaNodeAccessor { } leda_node;
struct LedaEdgeAccessor { } leda_edge;

template<class Iter>
node get_object(const LedaNodeAccessor&, const Iter& it) {
    return it.get_node(); }

template< class Iter>
edge get_object(const LedaEdgeAccessor&, const Iter& it) {
    return it.get_edge(); }
```

3. Operations

T get(*HandlerAccessor*<*T*, *Handler*, *StructAccessor*>*da*, *Iter it*)
returns the associated value of *it* for this accessor.

void set(*HandlerAccessor<T, Handler, StructAccessor>da, Iter it, T val*)
sets the associated value of *it* for this accessor to the given value.

4. Implementation

Constant Overhead.

5. Example

An exemplary application:

```
typedef HandlerAccessor<int,edge_array<int>,LedaEdgeAccessor> Length;
edge_array<int> edge_lengths(G);
Length          length(edge_lengths, leda_edge);
int c=0;
for(LedaLinEdgeIt it1(G); it1.valid(); ++it1, ++c)
    set(length,it1,c);
for(LedaLinNodeIt it2(G); it2.valid(); ++it2)
    for(LedaAdjIt it3(G,it2); it3.valid(); ++it3)
        cout << get(length,it3);
```

1.11 Member Accessors (MemberAccessor)

1. Definition

An instance *DA* of class *MemberAccessor<Str, StructAccessor, T>* manages the access to a node or edge parameter that is organized as a member of a struct, which is in some sense "attached" to the node or edge. The parameter is of type `T` and the struct of type `Str`. Given a handle or iterator, an object of class `StructAccessor` is required to allow access to the struct object, see below for the concrete syntactic requirements.

In particular, the concrete instantiation of `StructAccessor` determines whether the parameter is a node or edge parameter. The instance *DA* accesses its parameter through a pointer to member of type `Ptr`, which is defined by

```
typedef T Str::*Ptr
```

2. Creation

MemberAccessor<Str, StructAccessor, T> DA(StructAccessor sa, Ptr ptr);
creates an instance *DA* of this class, which is bound to *sa* and *ptr*.

The following definitions of `LedaMemberNodeAccessor` and `LedaMemberEdgeAccessor` fulfill the requirements stated above. They have been implemented and are available for immediate use with all handlers and iterators that provide a method `get_node` or `get_edge` like the methods provided by `NodeHandle<>` and `EdgeHandle<>`.

```
struct LedaMemberNodeAccessor {  } leda_member_node;
```

```
struct LedaMemberEdgeAccessor {  } leda_member_edge;

template<class SN, class Iter>
SN& get_object(const LedaMemberNodeAccessor&,const SN&, const Iter& it) {
   return (* LEDA-internals *); }

template<class SE, class Iter>
SE& get_object(const LedaMemberEdgeAccessor&, const SE&, const Iter& it) {
   return (* LEDA-internals *); }
```

T get(*MemberAccessor<Str, StructAccessor, T>ma, Iter it*)
returns the associated value of *it* for this accessor.

void set(*MemberAccessor<Str, StructAccessor, T>ma, Iter it, T val*)
sets the associated value of *it* for this accessor to the given value.

4. Implementation

Constant Overhead.

5. Example

Note that in the exemplary application of `HandlerAccessor` (Sect.1.10) only the definition of length must be changed:

```
typedef MemberAccessor<Str,LedaMemberEdgeAccessor,int> Length;
Length          length(leda_member_edge, &Str::length);
```

1.12 Method Accessors (MethodAccessor)

1. Definition

An instance *MA* of class *MethodAccessor<Str, StructAccessor, T>* will get as parameter the pointer to the method in the iterator. With every execution of `get()`, the value will be computed by an execution of that method. This class is very similar to the class `MemberAccessor`. The main point here is that not data members are addressed, but methods of classes. The given data accessor *da* is intented to yield for every iterator the struct (e.g. `str` of type `Str`), in which this method is located (`(str.*ptr)()` executes this function).

2. Creation

MethodAccessor<Str, StructAccessor, T> MA(StructAccessor sa, Ptr ptr);
creates an instance *MA* of this class, which is bound to *sa* and *ptr*.

3. Operations

T get(*MethodAccessor<Str, StructAccessor, T>ma, Iter it*)
returns the value that was computed by the execution of the method that was addressed by the internal method pointer.

4. Implementation

Constant overhead.

5. Example

Note that in the exemplary application of `HandlerAccessor` (Sect.1.10) only the definition of length must be changed:

```
typedef MethodAccessor<Str,LedaMemberEdgeAccessor,int> Length;
Length        length(leda_member_edge, &Str::length());
```

Another example would be an application of observers, `Obs` is an observer structure (defined earlier), `ObsAI` an observer iterator. We define a data accessor that yields for every observer iterator its observer (`ObsAccessor`). We assume that `Obs` has a method `length()`, that returns a computed length value.

```
template<class I> struct ObsAccessor { };
template<class Iter, class Obs>
Obs get_object(const ObsAccessor<Obs>&, const Iter& it) {
          return it.get_observer(); }
```

application of this class:

```
ObsAccessor<Obs>              obsacc;
MethodAccessor<Obs,ObsAccessor<Obs>,TE>  length2(obsacc,&Obs::length);
cout << get(length,ai);
```

The result will be, that every execution of `get` of this accessor will cause the execution of method `Obs::length`.

1.13 Constant Accessors (ConstantAccessor)

1. Definition

An instance *CA* of class *ConstantAccessor*<*T*> is bound to a specific value of type `T`, and the function `get()` simply returns this value for each iterator.

2. Creation

ConstantAccessor<*T*> *CA*(*T t*);
creates an instance *CA* of this class bound to the given value *t*.

3. Operations

T get(*ConstantAccessor*<*T*>*ca*, *Iter it*)
returns the value to which *CA* is bound.

1.14 Calculating Accessors (CalcAccessor)

1. Definition

Class `CalcAccessor` is meaningful for edge parameters only. Often, an edge parameter is determined by the incident nodes. For example, if the nodes represent points in the plane and an edge is a straight connection between its endpoints, the parameter "Euclidean length" is determined by the coordinates of the nodes.

An object *CA* of class *CalcAccessor*<*T*, *SourceAcc*, *TargetAcc*, *Calc*> can be instantiated by an algorithm that computes an edge parameter from information provided by the incident nodes.

2. Creation

CalcAccessor<*T*, *SourceAcc*, *TargetAcc*, *Calc*> *CA*(*SourceAcc sa*, *TargetAcc ta*, *Calc calc*);
creates an instance *CA* of this class.

Requirements:

- `T` is equal to the return type of `Calc`, i.e. `Calc::value_type` - this is necessary because of limitations of the used compiler
- `SourceAcc` data accessor that returns for every edge a user-defined structure associated with the source node of that edge
- `TargetAcc` data accessor that returns for every edge a user-defined structure of the target node
- `Calc` must fulfill the following requirements: The type `Calc::value_type` is defined (the return type of the calculation), and there is a template function

  ```
  calculate (Calc calc, Calc::value_type& result,
        SourceAcc::value_type source, TargetAcc::value_type target);
  ```

  which does the actual computation.

3. Operations

T get(*CalcAccessor*<*T*, *SourceAcc*, *TargetAcc*, *Calc*>*ca*, *Iter it*)
returns the value that was computed by the internal calculator.

4. Implementation

Asymptotic run time is dominated by `calculate()`.

5. Example

Assume that we have different data accessors:

- `X_coord` is a data accessor and provides access to the x-coordinate of a node that represents a point in the plane. For instance, if the first template parameter of the Leda class `GRAPH<>` is a struct of 2–dim coordinates, e.g. `GRAPH<PosStruct,anything>` with

```
struct PosStruct {
  double x,y;
};
```

then `X_coord` is a `MemberAccessor` (Sect. 1.10).

- `Y_coord` is defined analogously to `X_coord`.

Next we create a class that computes the Euclidean length of an edge from the coordinates of its endpoints:

```
template <class DA_X, class DA_Y>
struct EuclDistCalc {
   typedef double value_type;
   DA_X  X_coord;
   DA_Y  Y_coord;
};

template <class Calc, class T1, class T2, class Result>
void calculate (const Calc& calc, Result& result,
    T1 source, T2 target) {
   typedef Calc::value_type T;
   T x_diff = get(calc.X_coord,target) - get(calc.X_coord,source);
   T y_diff = get(calc.Y_coord,target) - get(calc.Y_coord,source);
   result= sqrt(x_diff*x_diff+y_diff*y_diff);
}
```

application of these classes (`Length_type` is return type of the calculation; `SA` and `TA` are instances of the data accessors `SourceAcc` and `TargetAcc`, respectively):

```
typedef EuclDistCalc<DA_X,DA_Y> Euclidean;
Euclidean       EC;
CalcAccessor<Length_type,SourceAcc,TargetAcc,Euclidean>
                eucl_dist(SA,TA,EC);

cout << "Euclidean Distance of source adjacency iterator: ";
cout << get(eucl_dist,ai) << endl;
```

1.15 Predicate Classes

A predicate class is templated by a handler / iterator type `Iter` and has a method

```
bool operator() (const Iter& );
```

This method defines a predicate on the set of all nodes or edges. Note that this is exactly the requirement for predicates in Sect. 1.7. However, predicates are useful in many other situations, too.

Preconditions:

- `Iterator` fulfills the same requirements as the iterator class used in the skip iterator template parameter list

- `StructAccessor` is a data accessor that provides a 'get'-function (e.g. Sect. 1.10, 1.11 or 1.12).
- `Compare` is a class that compares two items of the template parameter `T` by means of a method

  ```
  bool operator()(T,T);
  ```

  There are some classes available for this purpose: `Equal<T>`, `Unequal<T>`, `LessThan<T>`, `LessEqual<T>`, `GreaterThan<T>` and `GreaterEqual<T>` with obvious semantics, where `T` is the type of the values.

In the following example, a data accessor is firstly defined (`visible` of type `Visible`). Then, Predicate is defined with the data accessor as parameter. With this example, one may choose which nodes can be seen by the algorithm.

```
LedaNodeAccessor              NodeAcc;
node_array<bool>              values(G,true);
typedef HandlerAccessor<bool,node_array<bool>,LedaNodeAccessor> Visible;
Visible                       visible(values,NodeAcc);
typedef ComparePred<LedaAdjIt,Visible,Equal<bool> >             Predicate;
Predicate                     skip(visible,Equal<bool>(),true);
```

1.16 Example: Dijkstra

This section introduces an alternative, more flexible implementation of the Dijkstra algorithm, which can be adapted to various different needs. The basic concepts to achieve such a high degree of flexibility are the following:

- The algorithm does not work on a single data structure that represents the whole graph. Instead, it expects a couple of light–weight data structures, each of which allows access to a specific aspect of the graph.

  For this purpose it works on the Toolbox of Graph Algorithms (see Sect. 1.1).
- The algorithm is not implemented as a function, but as a class.

  This class provides a method to execute one iteration of the core loop of the Dijkstra algorithm. Hence, the algorithm can be stopped after each iteration. This makes it possible to change the algorithm's behavior (e.g. the break condition), to extend its functionality or even to intermix its execution with other algorithms.

For example, the following scenarios are covered (see the Examples clause below and Sects. 1.16.1–1.16.5).

Problem	Solution
compute a shortest path between two nodes s and t	use function `dijkstra_pair`
Euclidean case: compute a shortest (s,t)-path in a bounded area in the plane	`dijkstra_pair` with `SkipAdjIt` and a predicate that checks edges for being inside the area
operation counting—how many operations ?	observer in adjacency iterator that counts the number of executions; see example for observer iterators in Sect. 1.8
on-line checking - e.g. are the edge lengths bounded by a certain constant ?	can be done in data accessor for length that is checking the value in every call of function `get`
simultaneous graphical animation of the algorithm working	use observer iterators with an observer that has access to a special window in which all drawings take place
preemptive Dijkstra - the algorithm can be interrupted and continued again	use class `Dijkstra` and its member functions
pseudo-parallel execution of several instances of the Dijkstra algorithm (e.g. instances perform single iterations in a round-robin style)	use class `Dijkstra` and several instances of it; note that the use of `node_pq` is not possible, since there are several priority queues active at the same time
edge lengths and/or node distances are realized as template parameters of `GRAPH` instead of `edge_array` and `node_array`	use function `dijkstra` with instances of `MemberAccessor`
shortest paths of a certain node set $R \cup V$, i.e. find $\forall v \in V$ a minimum distance $\min\{dist(r,v) : r \in R\}$ that corresponds to that minimum distance	use class `Dijkstra` with R as set of sources.

1. Definition

An instance *algorithm* of class *Dijkstra<AdjIt, Length, Distance, PriorityQueue>* is an implementation of Dijkstra that can be flexibly initialized, stopped after each iteration of the core loop, and continued, time and again.

The requirements for the priority queue are ...

default constructor	initializes the queue to be empty
copy constructor	copies the queue
destructor	destructs the queue
`push(dist,it)`	pushes iterators with certain distances on the queue (`dist` is of type `Distance::value_type`)
`pop()`	pops iterator with smallest distance
`clear()`	clears the queue
`empty()`	true iff the queue is empty

2. Creation

Dijkstra<AdjIt, Length, Distance, PriorityQueue> algorithm(Length l, Distance d);

creates an instance *algorithm* of this class. The length and distance data accessors are initialized by the parameter list. The set of sources is empty.

The template parameter requirements are ...

- `AdjIt` adjacency iterator that iterates over the incident edges of a certain node
- `Length` read only data accessor that gives access to the length of edges

- `Distance` read/write data accessor that stores the distance of nodes

  Precondition: All edge lengths are initialized by values that are large enough to be taken as infinity.

  Remark: This precondition is not necessary for the algorithm to have a defined behavior. In fact, it may even make sense to break this precondition deliberately. For example, if the distances have been computed before and shall only be updated after inserting new edges, it makes perfect sense to start the algorithm with these distances.

- `PriorityQueue` priority queue with certain requirements

Precondition:

For a completely new computation, the node distances of all nodes are initialized to infinity (i.e. `Distance::max()`).

3. Operations

PriorityQueue& *algorithm*.queue( )		gives direct access to internal priority queue.
void	*algorithm*.init(*AdjIt s*)	s is added to the set of sources.
bool	*algorithm*.finished( )	is true iff the algorithm is finished, i.e. the priority queue is empty.
void	*algorithm*.next( )	performs one iteration of the core loop of the algorithm.
void	*algorithm*.finish_algo( )	executes the algorithm until *finished*( ) is true, i.e. exactly if the priority queue is empty.

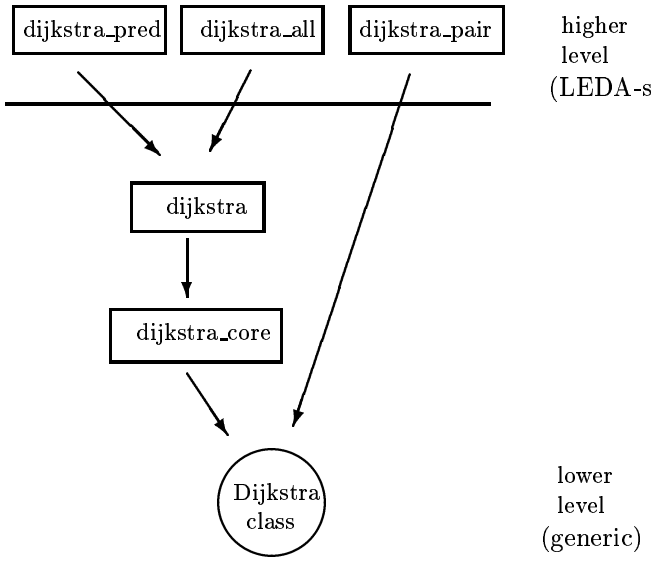
4. Implementation

The asymptotic complexity is $\mathcal{O}(m + n \cdot T(n))$, where $T(n)$ is the (possibly amortized) complexity of a single queue update.

For the priority queues described in Chapter 6, it is $T(n) = \mathcal{O}(\log n)$.

5. Example

The discessed functions are designed for typical applications of *Dijkstra*. They have been implemented and are available for instant use.



The restricted implementations are examples for how to customize the general implementations. In detail:

- Examples for the usage of function `dijkstra`: functions `dijkstra_pred` and `dijkstra_all`.
- Example for the usage of function `dijkstra_core`: function `dijkstra`.
- Examples for the usage of class `Dijkstra`: functions `dijkstra_pair` and `dijkstra_core`.

void dijkstra_core(*AdjIt s*, *Length length*, *Distance distance*, *PriorityQueue*& *pq*)

creates an instance *d* of the class *Dijkstra*<*AdjIt*, *Length*, *Distance*, *PriorityQueue*> with the parameter list. The algorithm will be started with *d.init*(*s*) and finished with *d.finish_algo*( ).
Precondition: All edge lengths are initialized by values that are large enough to be taken as infinity.

void dijkstra(*AdjIt s*, *LinNodeIt it*, *Length length*, *Distance distance*)

creates an instance *pq* of *LedaPQ* and initializes the distance value of the domain of the linear node iterator to the maximum value of the distance value type. Then it executes *dijkstra_core*(*s*, *length*, *distance*, *pq*).

Additional LEDA-style operations

The following functions are customized to the Leda graph structure and to node and edge arrays. This customization is hidden, and the usage of the functions is completely analogous to function `DIJKSTRA` (Chapter 7.20.2).

void dijkstra_pred(*graph G*, *node s*, *edge_array*<*TE*>*length*, *node_array*<*TN*>& *dist*, *node_array*<*edge*>& *pred*)

computes for a given graph *G* and edge lengths the shortest path distances and predecessors of shortest path tree.

void dijkstra_all(*graph G*, *node s*, *edge_array*<*TE*>*length*, *node_array*<*TN*>& *dist*)

computes for a given *G* and length datas for the edges the shortest path distances.

TE dijkstra_pair(*graph G*, *node s*, *node t*, *edge_array<TE>length*)

computes for a given graph G, nodes s and t a shortest (s,t)–path and returns its length.

TE dijkstra_pair(*graph G*, *node s*, *node t*, *edge_array<TE>length*, *list<edge>& path*)

computes for a given G , nodes s and t a shortest (s,t)–path and stores the predecessors.
postcondition: *path* contains all edges of a shortest path in ascending order from s to t.

Implementation of the functions

dijkstra_core( ), *dijkstra*( ), *dijkstra_pred*( ) and *dijkstra_all*( ) have the same time complexity as the LEDA-Version *DIJKSTRA*( ), i.e. $\mathcal{O}(m + n \cdot log\, n)$, where n is the number of nodes and m the number of edges in the graph.

1.17 Source code of the functions

1.17.1 function dijkstra_core

```
template<class AdjIt,class Length,class Distance,class PriorityQueue>
void dijkstra_core(AdjIt s, const Length& length, Distance& distance,
                   PriorityQueue& pq) {
  Dijkstra<AdjIt,Length,Distance,PriorityQueue>
               internal_dijk(length,distance);
  internal_dijk.queue()=pq;
  set(distance,s,Distance::null());
  if (s.valid()) {
    internal_dijk.init(s);
    internal_dijk.finish_algo();
  }
}
```

1.17.2 function dijkstra

```
template<class AdjIt,class LinNodeIt,class Length,class Distance>
void dijkstra(AdjIt s, LinNodeIt it, const Length& length, Distance& distance) {
  LedaPQ<Distance::value_type,AdjIt> pq;
  while(it.valid()) {
        set(distance,it,Distance::max());
        ++it; }
  dijkstra_core(s, length, distance, pq);
}
```

1.17.3 function dijkstra_all

```
template<class TN, class TE>
```

```
void dijkstra_all(const graph & G, node s,
        const edge_array<TE>& length,
        node_array<TN>& dist ) {
  LedaAdjIt         ai(G,s);
  LedaLinNodeIt     it(G);
  typedef HandlerAccessor<TN, node_array<TN>, LedaNodeAccessor> Distance;
  Distance distance(dist,leda_node)
  typedef HandlerAccessor<TE, edge_array<TE>, LedaEdgeAccessor>  Length;
  Length length2(length,leda_edge);
  dijkstra(ai, it, length2, distance);
}
```

1.17.4 function dijkstra_pred

```
template<class TN, class TE>
void dijkstra_pred(const graph& G, node s,
        const edge_array<TE> & length,
        node_array<TN>& dist,
        node_array<edge>& pred ) {
  LedaAdjIt               ai(G,s);
  LedaLinNodeIt           it(G);
  typedef PredAccessor<TN, node_array<TN> >     Distance;
  typedef LenAccessor<TE, edge_array<TE> >      Length;
  edge pred_container;
  Distance distance(pred_container,dist,pred);
  Length length2(pred_container,length);
  dijkstra(ai, it, length2, distance);
}
```

1.17.5 function dijkstra_pair

```
template<class TE>
TE dijkstra_pair(const graph& G, node s, node t,
        const edge_array<TE> & length) {
  typedef TE                        TN;
  typedef LedaPQ<TN,LedaAdjIt>    PriorityQueue;
  node_array<TN>                    dist(G);
  if (s==t) return 0;
  (* initialization-part of dijkstra_all repeated *)
  PriorityQueue pq;
  Dijkstra<LedaAdjIt, Length, Distance, PriorityQueue>
               internal_dijk(length2,distance);
  while(it.valid()) {
        dist[it.get_node()]=Distance::max();
        ++it; }
  if (ai.eol()) return Distance::max();
  internal_dijk.queue()=pq;
  dist[s]=0;
  internal_dijk.init(ai);
```

```
  while(dist[t]==Distance::max() && !(internal_dijk.queue().empty()))
  internal_dijk.next();
  return dist[t];
}
```

1.17.6 function dijkstra_pair (version 2)

```
template<class TE>
TE dijkstra_pair(const graph& G, node s, node t,
        const edge_array<TE> & length,
        list<edge>& path) {
  typedef int                      TN;
  typedef LedaPQ<TN,LedaAdjIt>     PriorityQueue;
  node_array<TN>                   dist(G);
  node_array<edge>                 pred(G);
  if (s==t) return 0;
  (* initialization-part of dijkstra_pred repeated *)
  PriorityQueue pq;
  path.clear();
  Dijkstra<LedaAdjIt, Length, Distance, PriorityQueue>
        internal_dijk(length2,distance);
  while(it.valid()) {
        dist[it.get_node()]=Distance::max();
        pred[it.get_node()]=nil;
        ++it; }
  if (ai.eol()) return Distance::max();
  internal_dijk.queue()=pq;
  dist[s]=0;
  internal_dijk.init(ai);
  while(pred[t]==nil && !(internal_dijk.queue().empty()))
     internal_dijk.next();
  if (dist[t]<Distance::max()) {
    edge e=pred[t];
    while(pred[source(e)]!=nil) {
          path.push(e);
          e=pred[source(e)]; }
  }
  return dist[t];
}
```