Chapter 9

Graphs and Related Data Types

9.1 Graphs (graph)

1. Definition

An instance G of the data type *graph* consists of a list V of nodes and a list E of edges (*node* and *edge* are item types). Distinct graphs have disjoint node and edge lists. The value of a variable of type node is either the node of some graph, or the special value *nil* (which is distinct from all nodes), or is undefined (before the first assignment to the variable). A corresponding statement is true for the variables of type edge.

A graph with empty node list is called *empty*. A pair of nodes $(v, w) \in V \times V$ is associated with every edge $e \in E$; v is called the *source* of e and w is called the *target* of e, and v and w are called *endpoints* of e. The edge e is said to be *incident* to its endpoints.

A graph is either *directed* or *undirected*. The difference between directed and undirected graphs is the way the edges incident to a node are stored and how the concept *adjacent* is defined.

In directed graphs two lists of edges are associated with every node v: $adj\_edges(v) = \{e \in E \mid v = source(e)\}$, i.e., the list of edges starting in v, and $in\_edges(v) = \{e \in E \mid v = target(e)\}$, i.e., the list of edges ending in v. The list $adj\_edges(v)$ is called the adjacency list of node v and the edges in $adj\_edges(v)$ are called the edges *adjacent* to node v. For directed graphs we often use $out\_edges(v)$ as a synonym for $adj\_edges(v)$.

In undirected graphs only the list $adj\_edges(v)$ is defined for every every node v. Here it contains all edges incident to v, i.e., $adj\_edges(v) = \{e \in E \mid v \in \{source(e), target(e)\}\}$. An undirected graph may not contain selfloops, i.e., it may not contain an edge whose source is equal to its target.

In a directed graph an edge is adjacent to its source and in an undirected graph it is adjacent to its source and target. In a directed graph a node w is adjacent to a node v if

there is an edge $(v, w) \in E$; in an undirected graph w is adjacent to v if there is an edge (v, w) or (w, v) in the graph.

A directed graph can be made undirected and vice versa: G.make_undirected() makes the directed graph G undirected by appending for each node v the list $in\_edges(v)$ to the list $adj\_edges(v)$ (removing selfloops). Conversely, G.make_directed() makes the undirected graph G directed by splitting for each node v the list $adj\_edges(v)$ into the lists $out\_edges(v)$ and $in\_edges(v)$. Note that these two operations are not exactly inverse to each other. The data type *ugraph* (cf. section 9.4) can only represent undirected graphs.

Reversal Information, Maps and Faces

The reversal information of an edge e is accessed through $G.reversal(e)$, it has type *edge* and may or may not be defined ($= nil$). Assume that $G.reversal(e)$ is defined and let $e' = G.reversal(e)$. Then $e = (v, w)$ and $e' = (w, v)$ for some nodes v and w, $G.reversal(e')$ is defined and $e = G.reversal(e')$. In addtion, $e \neq e'$. In other words, *reversal* deserves its name.

We call a directed graph *bidirected* if the reversal information can be properly defined for all edges in G, resp. if there exists a bijective function $rev : E \to E$ with the properties of *reversal* as described above and we call a bidirected graph a *map* if all edges have their reversal information defined. Maps are the data structure of choice for embedded graphs. For an edge e of a map G let $face\_cycle\_succ(e) = cyclic\_adj\_pred(reversal(e))$ and consider the sequence e, $face\_cycle\_succ(e)$, $face\_cycle\_succ(face\_cycle\_succ(e))$, ... The first edge to repeat in this sequence is e (why?) and the set of edges appearing in this sequence is called the *face cycle* containing e. Each edge is contained in some face cycle and face cycles are pairwise disjoint. Let f be the number of face cycles, n be the number of (non-isolated) nodes, m be the number of edges, and let c be the number of (non-singleton) connected components. Then $g = (m/2 - n - f)/2 + c$ is called the *genus* of the map [82] (note that $m/2$ is the number of edges in the underlying undirected graph). The genus is zero if and only if the map is planar, i.e., there is an embedding of G into the plane such that for every node v the counter-clockwise ordering of the edges around v agrees with the cyclic ordering of v's adjacency list. (In order to check whether a map is planar, you may use the function *Is_Plane_Map*( ) in 9.23.)

If a graph G is a map the faces of G can be constructed explicitly by $G.compute\_faces(\ )$. Afterwards, the faces of G can be traversed by different iterators, e.g., $forall\_faces(f, G)$ iterates over all faces , $forall\_adj\_faces(v)$ iterates over all faces adjacent to node v. By using face maps or arrays (data types *face_map* and *face_array*) additional information can be associated with the faces of a graph. Note that any update operation performed on G invalidates the list of faces. See the section on face operations for a complete list of available operations for faces.

#include < *LEDA/graph.h* >

2. Creation

graph G; creates an object G of type *graph* and initializes it to the empty directed graph.

graph G(*int n_slots*, *int e_slots*);

this constructor specifies the numbers of free data slots in the nodes and edges of G that can be used for storing the entries of node and edge arrays. See also the description of the *use_node_data*( ) and *use_edge_data*( ) operations in 9.8 and 9.9.

3. Operations

void	G.init(*int* n, *int* m)	this operation has to be called for semi-dynamic graphs (if compiled with $-DGRAPH\_REP = 2$) immediately after the constructor to specify upper bounds n and m for the number of nodes and edges respectively. This operation has no effect if called for the (fully-dynamic) standard graph representation.

a) Access operations

int	G.outdeg(*node* v)	returns the number of edges adjacent to node v ($\|adj\_edges(v)\|$).
int	G.indeg(*node* v)	returns the number of edges ending at v ($\|in\_edges(v)\|$) if G is directed and zero if G is undirected).
int	G.degree(*node* v)	returns $outdeg(v) + indeg(v)$.
node	G.source(*edge* e)	returns the source node of edge e.
node	G.target(*edge* e)	returns the target node of edge e.
node	G.opposite(*node* v, *edge* e)	returns $target(e)$ if $v = source(e)$ and $source(e)$ otherwise.
node	G.opposite(*edge* e, *node* v)	same as above.
int	G.number_of_nodes( )	returns the number of nodes in G.
int	G.number_of_edges( )	returns the number of edges in G.
const list<node>&	G.all_nodes( )	returns the list V of all nodes of G.
const list<edge>&	G.all_edges( )	returns the list E of all edges of G.

list<*edge*>	G.adj_edges(*node* v)	returns $adj\_edges(v)$.
list<*edge*>	G.out_edges(*node* v)	returns $adj\_edges(v)$ if G is directed and the empty list otherwise.
list<*edge*>	G.in_edges(*node* v)	returns $in\_edges(v)$ if G is directed and the empty list otherwise.
list<*node*>	G.adj_nodes(*node* v)	returns the list of all nodes adjacent to v.
node	G.first_node( )	returns the first node in V.
node	G.last_node( )	returns the last node in V.
node	G.choose_node( )	returns a random node of G (nil if G is empty).
node	G.succ_node(*node* v)	returns the successor of node v in V (nil if it does not exist).
node	G.pred_node(*node* v)	returns the predecessor of node v in V (nil if it does not exist).
edge	G.first_edge( )	returns the first edge in E.
edge	G.last_edge( )	returns the last edge in E.
edge	G.choose_edge( )	returns a random edge of G (nil if G is empty).
edge	G.succ_edge(*edge* e)	returns the successor of edge e in E (nil if it does not exist).
edge	G.pred_edge(*edge* e)	returns the predecessor of edge e in E (nil if it does not exist).
edge	G.first_adj_edge(*node* v)	returns the first edge in the adjacency list of v (nil if this list is empty).
edge	G.last_adj_edge(*node* v)	returns the last edge in the adjacency list of v (nil if this list is empty).
edge	G.adj_succ(*edge* e)	returns the successor of edge e in the adjacency list of node $source(e)$ (nil if it does not exist).
edge	G.adj_pred(*edge* e)	returns the predecessor of edge e in the adjacency list of node $source(e)$ (nil if it does not exist).
edge	G.cyclic_adj_succ(*edge* e)	returns the cyclic successor of edge e in the adjacency list of node $source(e)$.
edge	G.cyclic_adj_pred(*edge* e)	returns the cyclic predecessor of edge e in the adjacency list of node $source(e)$.

edge	*G*.first_in_edge(*node v*)	returns the first edge of $in\_edges(v)$ (nil if this list is empty).
edge	*G*.last_in_edge(*node v*)	returns the last edge of $in\_edges(v)$ (nil if this list is empty).
edge	*G*.in_succ(*edge e*)	returns the successor of edge e in $in\_edges(target(e))$ (nil if it does not exist).
edge	*G*.in_pred(*edge e*)	returns the predecessor of edge e in $in\_edges(target(e))$ (nil if it does not exist).
edge	*G*.cyclic_in_succ(*edge e*)	returns the cyclic successor of edge e in $in\_edges(target(e))$ (nil if it does not exist).
edge	*G*.cyclic_in_pred(*edge e*)	returns the cyclic predecessor of edge e in $in\_edges(target(e))$ (nil if it does not exist).
bool	*G*.is_directed( )	returns true iff G is directed.
bool	*G*.is_undirected( )	returns true iff G is undirected.
bool	*G*.empty( )	returns true iff G is empty.

b) Update operations

node	*G*.new_node( )	adds a new node to G and returns it.
node	*G*.new_node(*node u*, *int dir*)	adds a new node v to G and returns it. v is inserted before ($dir = leda::before$) or after ($dir = leda::after$) node u into the list of all nodes.
edge	*G*.new_edge(*node v*, *node w*)	adds a new edge (v, w) to G by appending it to $adj\_edges(v)$ and to $in\_edges(w)$ (if G is directed) or $adj\_edges(w)$ (if G is undirected), and returns it.
edge	*G*.new_edge(*edge e*, *node w*, *int dir* = *leda*::*after*)	adds a new edge $x = (source(e), w)$ to G. x is inserted before ($dir = leda::before$) or after ($dir = leda::after$) edge e into $adj\_edges(source(e))$ and appended to $in\_edges(w)$ (if G is directed) or $adj\_edges(w)$ (if G is undirected). Here $leda::before$ and $leda::after$ are predefined constants. The operation returns the new edge x. *Precondition*: $source(e) \neq w$ if G is undirected.

edge G.new_edge(*node* v, *edge* e, *int* $dir = leda::after$)

adds a new edge $x = (v, target(e))$ to G. x is appended to $adj\_edges(v)$ and inserted before ($dir = leda::before$) or after ($dir = leda::after$) edge e into $in\_edges(target(e))$ (if G is directed) or $adj\_edges(target(e))$ (if G is undirected). The operation returns the new edge x.
Precondition: $target(e) \neq v$ if G is undirected.

edge G.new_edge(*edge e1*, *edge e2*, *int d1* = $leda::after$, *int d2* = $leda::after$)

adds a new edge $x = (source(e1), target(e2))$ to G. x is inserted before (if $d1 = leda::before$) or after (if $d1 = leda::after$) edge $e1$ into $adj\_edges(source(e1))$ and before (if $d2 = leda::before$) or after (if $d2 = leda::after$) edge $e2$ into $in\_edges(target(e2))$ (if G is directed) or $adj\_edges(target(e2))$ (if G is undirected). The operation returns the new edge x.

node G.merge_nodes(*node v1*, *node v2*)

experimental.

node G.merge_nodes(*edge e1*, *node v2*)

experimental.

node G.split_edge(*edge e*, *edge*& *e1*, *edge*& *e2*)

experimental

void G.hide_edge(*edge e*) removes edge e temporarily from G until restored by $G.restore\_edge(e)$.

bool G.is_hidden(*edge e*) returns *true* if e is hidden and *false* otherwise.

list<*edge*> G.hidden_edges( ) returns the list of all hidden edges of G.

void G.restore_edge(*edge e*) restores e by appending it to $adj\_edges(source(e))$ and to $in\_edges(target(e))$ ($adj\_edges(target(e))$ if G is undirected). *Precondition*: e is hidden and neither $source(e)$ nor $target(e)$ is hidden.

void G.restore_all_edges( ) restores all hidden edges.

void G.hide_node(*node v*) removes node v temporarily from G until restored by $G.restore\_node(v)$. All non-hidden edges in $adj\_edges(v)$ and $in\_edges(v)$ are hidden too.

void G.hide_node(*node v*, *list*<*edge*>& *h_edges*)

as above, in addition, the list of leaving or entering edges which are hidden as a result of hiding v are appended to *h_edges*.

bool G.is_hidden(*node v*) returns *true* if v is hidden and *false* otherwise.

list<*node*> G.hidden_nodes( ) returns the list of all hidden nodes of G.

void G.restore_node(*node v*) restores v by appending it to the list of all nodes. Note that no edge adjacent to v that was hidden by $G.hide\_node(v)$ is restored by this operation.

void G.restore_all_nodes( ) restores all hidden nodes.

void G.del_node(*node v*) deletes v and all edges incident to v from G.

void G.del_edge(*edge e*) deletes the edge e from G.

void G.del_nodes(*const list*<*node*>& L)

deletes all nodes in L from G.

void G.del_edges(*const list*<*edge*>& L)

deletes all edges in L from G.

void G.del_all_nodes( ) deletes all nodes from G.

void G.del_all_edges( ) deletes all edges from G.

void G.del_all_faces( ) deletes all faces from G.

void G.move_edge(*edge e*, *node v*, *node w*)

moves edge e to source v and target w by appending it to $adj\_edges(v)$ and to $in\_edges(w)$ (if G is directed) or $adj\_edges(w)$) (if G is undirected).

void G.move_edge(*edge e*, *edge e1*, *node w*, *int d* = *leda*::*after*)

moves edge e to source $source(e1)$ and target w by inserting it before (if $d = leda::before$) or after (if $d = leda::after$) edge $e1$ into $adj\_edges(source(e1))$ and by appending it to $in\_edges(w)$ (if G is directed) or $adj\_edges(w)$) (if G is undirected).

void G.move_edge(*edge e*, *node v*, *edge e2*, *int d* = *leda*::*after*)

moves edge e to source v and target $target(e2)$ by appending it to $adj\_edges(v)$) and inserting it before (if $d = leda::before$) or after (if $d = leda::after$) edge $e2$ into $in\_edges(target(e2))$ (if G is directed) or $adj\_edges(target(e2))$ (if G is undirected).

void G.move_edge(*edge e*, *edge e1*, *edge e2*, *int d1* = *leda*::*after*, *int d2* = *leda*::*after*)

moves edge e to source $source(e1)$ and target $target(e2)$ by inserting it before (if *d1* = *leda*::*before*) or after (if *d1* = *leda*::*after*) edge $e1$ into $adj\_edges(source(e1))$ and before (if *d2* = *leda*::*before*) or after (if *d2* = *leda*::*after*) edge $e2$ into $in\_edges(target(e2))$ (if G is directed) or $adj\_edges(target(e2))$ (if G is undirected).

edge G.rev_edge(*edge e*) reverses e ($move\_edge(e, target(e), source(e))$).

void G.rev_all_edges( ) reverses all edges of G.

void G.sort_nodes(*int* (∗*cmp*)(*const node*& , *const node*&))

the nodes of G are sorted according to the ordering defined by the comparing function *cmp*. Subsequent executions of forall_nodes step through the nodes in this order. (cf. TOPSORT1 in section 10).

void G.sort_edges(*int* (∗*cmp*)(*const edge*& , *const edge*&))

the edges of G and all adjacency lists are sorted according to the ordering defined by the comparing function *cmp*. Subsequent executions of forall_edges step through the edges in this order. (cf. TOPSORT1 in section 10).

void G.sort_nodes(*const node_array*<*T*>& *A*)

the nodes of G are sorted according to the entries of node_array A (cf. section 9.8).
Precondition: T must be numerical.

void G.sort_edges(*const edge_array*<*T*>& *A*)

the edges of G are sorted according to the entries of edge_array A (cf. section 9.9).
Precondition: T must be numerical.

void G.bucket_sort_nodes(*int l*, *int h*, *int* (∗*ord*)(*const node*&))

sorts the nodes of G using *bucket sort*
Precondition: $l \leq ord(v) \leq h$ for all nodes v.

void G.bucket_sort_edges(*int l*, *int h*, *int* (∗*ord*)(*const edge*&))

sorts the edges of G using *bucket sort*
Precondition: $l \leq ord(e) \leq h$ for all edges e.

void G.bucket_sort_nodes(*int* (∗*ord*)(*const node*&))

same as $G.bucket\_sort\_nodes(l, h, ord$ with l (h) equal to the minimal (maximal) value of $ord(v)$.

void G.bucket_sort_edges(*int* (∗*ord*)(*const edge*&))

same as $G.bucket\_sort\_edges(l, h, ord$ with l (h) equal to the minimal (maximal) value of $ord(e)$.

void G.bucket_sort_nodes(*const node_array*<*int*>& A)

same as $G.bucket\_sort\_nodes(ord)$ with $ord(v) = A[v]$ for all nodes v of G.

void G.bucket_sort_edges(*const edge_array*<*int*>& A)

same as $G.bucket\_sort\_edges(ord)$ with $ord(e) = A[e]$ for all edges e of G.

void G.set_node_position(*node* v, *node* p)

moves node v in the list V of all nodes such that p becomes the predecessor of v. If $p = nil$ then v is moved to the front of V.

void G.set_edge_position(*edge* e, *edge* p)

moves edge e in the list E of all edges such that p becomes the predecessor of e. If $p = nil$ then e is moved to the front of E.

list<*edge*> G.insert_reverse_edges( ) for every edge (v, w) in G the reverse edge (w, v) is inserted into G. Returns the list of all inserted edges.
Remark: the reversal information is not set by this function.

void G.make_undirected( ) makes G undirected by appending $in\_edges(v)$ to $adj\_edges(v)$ for all nodes v.

void G.make_directed( ) makes G directed by splitting $adj\_edges(v)$ into $out\_edges(v)$ and $in\_edges(v)$.

void G.clear( ) makes G the empty graph.

void G.join(*graph*& H) merges H into G by moving all objects (nodes,edges, and faces) from H to G. H is empty afterwards.

c) Reversal Edges and Maps

void G.make_bidirected( ) makes G bidirected by inserting missing reversal edges.

void G.make_bidirected(*list*<*edge*>& R)

makes G bidirected by inserting missing reversal edges. Appends all inserted edges to list R.

bool G.is_bidirected( ) returns true if every edge has a reversal and false otherwise.

bool	*G*.make_map( )	sets the reversal information of a maximal number of edges of G. Returns *true* if G is bidirected and *false* otherwise.
void	*G*.make_map(*list*<*edge*>& R)	makes G bidirected by inserting missing reversal edges and then turns it into a map setting the reversals for all edges. Appends all inserted edges to list R.
bool	*G*.is_map( )	tests whether G is a map.
edge	*G*.reversal(*edge* e)	returns the reversal information of edge e (*nil* if not defined).
void	*G*.set_reversal(*edge* e, *edge* r)	makes r the reversal of e and vice versa. If the reversal information of e was defined prior to the operation, say as e', the reversal information of e' is set to nil. The same holds for r. *Precondition*: $e = (v, w)$ and $r = (w, v)$ for some nodes v and w.
edge	*G*.face_cycle_succ(*edge* e)	returns the cyclic adjacency predecessor of $reversal(e)$. *Precondition*: $reversal(e)$ is defined.
edge	*G*.face_cycle_pred(*edge* e)	returns the reversal of the cyclic adjacency successor s of e. *Precondition*: $reversal(s)$ is defined.
edge	*G*.split_map_edge(*edge* e)	splits edge $e = (v, w)$ and its reversal $r = (w, v)$ into edges (v, u), (u, w), (w, u), and (u, v). Returns the edge (u, w).
edge	*G*.new_map_edge(*edge* *e1*, *edge* *e2*)	inserts a new edge $e = (source(e1), source(e2)$ after $e1$ into the adjacency list of $source(e1)$ and an edge r reversal to e after $e2$ into the adjacency list of $source(e2)$.
list<*edge*>	*G*.triangulate_map( )	triangulates the map G by inserting additional edges. The list of inserted edges is returned. *Precondition*: G must be connected. The algorithm ([45]) has running time $O(\|V\| + \|E\|)$.
void	*G*.dual_map(*graph*& D)	constructs the dual of G in D. The algorithm has linear running time. *Precondition*: G must be a map.

For backward compatibility

edge	G.reverse(*edge* e)	returns *reversal*(e) (historical).
edge	G.succ_face_edge(*edge* e)	returns *face_cycle_succ*(e) (historical).
edge	G.next_face_edge(*edge* e)	returns *face_cycle_succ*(e) (historical).
edge	G.pred_face_edge(*edge* e)	returns *face_cycle_pred*(e) (historical).

d) Faces and Planar Maps

void	G.compute_faces( )	constructs the list of face cycles of G. *Precondition*: G is a map.
face	G.face_of(*edge* e)	returns the face of G to the left of edge e.
face	G.adj_face(*edge* e)	returns G.*face_of*(e).
void	G.print_face(*face* f)	prints face f.
int	G.number_of_faces( )	returns the number of faces of G.
face	G.first_face( )	returns the first face of G. (nil if empty).
face	G.last_face( )	returns the last face of G.
face	G.choose_face( )	returns a random face of G (nil if G is empty).
face	G.succ_face(*face* f)	returns the successor of face f in the face list of G (nil if it does not exist).
face	G.pred_face(*face* f)	returns the predecessor of face f in the face list of G (nil if it does not exist).
const list<*face*>&	G.all_faces( )	returns the list of all faces of G.
list<*face*>	G.adj_faces(*node* v)	returns the list of all faces of G adjacent to node v in counter-clockwise order.
list<*node*>	G.adj_nodes(*face* f)	returns the list of all nodes of G adjacent to face f in counter-clockwise order.
list<*edge*>	G.adj_edges(*face*)	returns the list of all edges of G bounding face f in counter-clockwise order.
int	G.size(*face* f)	returns the number of edges bounding face f.
edge	G.first_face_edge(*face* f)	returns the first edge of face f in G.

edge G.split_face(*edge e1*, *edge e2*)

inserts the edge $e = (source(e_1), source(e_2))$ and its reversal into G and returns e.
Precondition: e_1 and e_2 are bounding the same face F.
The operation splits F into two new faces.

face G.join_faces(*edge e*) deletes edge e and its reversal r and updates the list of faces accordingly. The function returns a face that is affected by the operations (see the LEDA book for details).

void G.make_planar_map( ) makes G a planar map by reordering the edges such that for every node v the ordering of the edges in the adjacency list of v corresponds to the counter-clockwise ordering of these edges around v for some planar embedding of G and constructs the list of faces.
Precondition: G is a planar bidirected graph (map).

list<*edge*> G.triangulate_planar_map( )

triangulates planar map G and recomputes its list of faces

e) Operations for undirected graphs

edge G.new_edge(*node v*, *edge e1*, *node w*, *edge e2*, *int d1* = *leda*::*after*, *int d2* = *leda*::*after*)

adds a new edge (v, w) to G by inserting it before (if *d1* = *leda*::*before*) or after (if *d1* = *leda*::*after*) edge $e1$ into $adj\_edges(v)$ and before (if *d2* = *leda*::*before*) or after (if *d2* = *leda*::*after*) edge $e2$ into $adj\_edges(w)$, and returns it.
Precondition: $e1$ is incident to v and $e2$ is incident to w and $v \neq w$.

edge G.new_edge(*node v*, *edge e*, *node w*, *int d* = *leda*::*after*)

adds a new edge (v, w) to G by inserting it before (if d = *leda*::*before*) or after (if d = *leda*::*after*) edge e into $adj\_edges(v)$ and appending it to $adj\_edges(w)$, and returns it.
Precondition: e is incident to v and $v \neq w$.

edge G.new_edge(*node* v, *node* w, *edge* e, *int* $d = leda::after$)

adds a new edge (v, w) to G by appending it to to $adj\_edges(v)$, and by inserting it before (if $d = leda::before$) or after (if $d = leda::after$) edge e into $adj\_edges(w)$, and returns it.
Precondition: e is incident to w and $v \neq w$.

edge G.adj_succ(*edge* e, *node* v)

returns the successor of edge e in the adjacency list of v.
Precondition: e is incident to v.

edge G.adj_pred(*edge* e, *node* v)

returns the predecessor of edge e in the adjacency list of v.
Precondition: e is incident to v.

edge G.cyclic_adj_succ(*edge* e, *node* v)

returns the cyclic successor of edge e in the adjacency list of v.
Precondition: e is incident to v.

edge G.cyclic_adj_pred(*edge* e, *node* v)

returns the cyclic predecessor of edge e in the adjacency list of v.
Precondition: e is incident to v.

f) I/O Operations

void G.write(*ostream*& O = *cout*)

writes G to the output stream O.

void G.write(*string* s) writes G to the file with name s.

int G.read(*istream*& I = *cin*)

reads a graph from the input stream I and assigns it to G.

int G.read(*string* s) reads a graph from the file with name s and assigns it to G. Returns 1 if file s does not exist, 2 if the edge and node parameter types of $*this$ and the graph in the file s do not match, 3 if file s does not contain a graph, and 0 otherwise.

bool G.write_gml(*ostream*& O = *cout*, *void* (∗*node_cb*)(*ostream*& , *const graph*∗, *const node*) = 0, *void* (∗*edge_cb*)(*ostream*& , *const graph*∗, *const edge*) = 0)

writes G to the output stream O in GML format ([44]). If *node_cb* is not equal to 0, it is called while writing a node v with output stream O, the graph and v as parameters. It can be used to write additional user defined node data. The output should conform with GML format (see manual page *gml_graph*). *edge_cb* is called while writing edges. If the operation fails, $false$ is returned.

bool G.write_gml(*string* s, *void* (∗*node_cb*)(*ostream*& , *const graph*∗, *const node*) = 0, *void* (∗*edge_cb*)(*ostream*& , *const graph*∗, *const edge*) = 0)

writes G to the file with name s in GML format. For a description of *node_cb* and *edge_cb*, see above. If the operation fails, $false$ is returned.

bool G.read_gml(*string* s)

reads a graph in GML format from the file with name s and assigns it to G.

bool G.read_gml(*istream*& I = *cin*)

reads a graph in GML format from the input stream I and assigns it to G.

void G.print_node(*node* v, *ostream*& O = *cout*)

prints node v on the output stream O.

void G.print_edge(*edge* e, *ostream*& O = *cout*)

prints edge e on the output stream O. If G is directed e is represented by an arrow pointing from source to target. If G is undirected e is printed as an undirected line segment.

void G.print(*string* s, *ostream*& O = *cout*)

prints G with header line s on the output stream O.

void G.print(*ostream*& O = *cout*)

prints G on the output stream O.

g) Non-Member Functions

node source(*edge* e)

returns the source node of edge e.

node target(*edge* e)

returns the target node of edge e.

graph∗ graph_of(*node* v)

returns a pointer to the graph that v belongs to.

graph∗	graph_of(*edge* e)	returns a pointer to the graph that e belongs to.
graph∗	graph_of(*face* f)	returns a pointer to the graph that f belongs to.
face	face_of(*edge* e)	returns the face of edge e.

h) Iteration

All iteration macros listed in this section traverse the corresponding node and edge lists of the graph, i.e. they visit nodes and edges in the order in which they are stored in these lists.

forall_nodes(v, G)
{ "the nodes of G are successively assigned to v" }

forall_edges(e, G)
{ "the edges of G are successively assigned to e" }

forall_rev_nodes(v, G)
{ "the nodes of G are successively assigned to v in reverse order" }

forall_rev_edges(e, G)
{ "the edges of G are successively assigned to e in reverse order" }

forall_adj_edges(e, w)
{ "the edges adjacent to node w are successively assigned to e" }

forall_out_edges(e, w) a faster version of **forall_adj_edges** for directed graphs.

forall_in_edges(e, w)
{ "the edges of $in\_edges(w)$ are successively assigned to e" }

forall_inout_edges(e, w)
{ "the edges of $adj\_edges(w)$ and $in\_edges(w)$ are successively assigned to e" }

forall_adj_nodes(v, w)
{ "the nodes adjacent to node w are successively assigned to v" }

Faces

Before using any of the following face iterators the list of faces has to be computed by calling $G.compute\_faces(\ )$. Note, that any update operation invalidates this list.

forall_faces(f, M)
{ "the faces of M are successively assigned to f" }

forall_face_edges(e, f)
{ "the edges of face f are successively assigned to e" }

forall_adj_faces(f, v)
{ "the faces adjacent to node v are successively assigned to f" }

4. Implementation

Graphs are implemented by doubly linked lists of nodes and edges. Most operations take constant time, except for all_nodes, all_edges, del_all_nodes, del_all_edges, make_map, make_planar_map, compute_faces, all_faces, make_map, clear, write, and read which take time $O(n + m)$, and adj_edges, adj_nodes, out_edges, in_edges, and adj_faces which take time $O(output\ size)$ where n is the current number of nodes and m is the current number of edges. The space requirement is $O(n + m)$.

9.2 Parameterized Graphs (GRAPH)

1. Definition

A parameterized graph G is a graph whose nodes and edges contain additional (user defined) data. Every node contains an element of a data type *vtype*, called the node type of G and every edge contains an element of a data type *etype* called the edge type of G. We use $<v, w, y>$ to denote an edge (v, w) with information y and $<x>$ to denote a node with information x.

All operations defined for the basic graph type *graph* are also defined on instances of any parameterized graph type *GRAPH*<*vtype*, *etype*>. For parameterized graphs there are additional operations to access or update the information associated with its nodes and edges. Instances of a parameterized graph type can be used wherever an instance of the data type *graph* can be used, e.g., in assignments and as arguments to functions with formal parameters of type *graph*&. If a function f(*graph*& G) is called with an argument Q of type *GRAPH*<*vtype*, *etype*> then inside f only the basic graph structure of Q can be accessed. The node and edge entries are hidden. This allows the design of generic graph algorithms, i.e., algorithms accepting instances of any parametrized graph type as argument.

#include < *LEDA/graph.h* >

2. Types

GRAPH<*vtype*, *etype*>:: *node_value_type*
the type of node data (*vtype*).

GRAPH<*vtype*, *etype*>:: *edge_value_type*
the type of edge data (*etype*).

3. Creation

GRAPH<*vtype*, *etype*> G; creates an instance G of type *GRAPH*<*vtype*, *etype*> and initializes it to the empty graph.

4. Operations

const vtype& G.inf(*node* v) returns the information of node v.

const vtype& G[*node* v] returns a reference to G.inf(v).

const etype& G.inf(*edge* e) returns the information of edge e.

const etype& G[*edge* e] returns a reference to G.inf(e).

node_array<*vtype*>& G.node_data( )

makes the information associated with the nodes of G available as a node array of type *node_array*<*vtype*>.

edge_array<*etype*>& G.edge_data( )

makes the information associated with the edges of G available as an edge array of type *edge_array*<*etype*>.

void G.assign(*node* v, *const vtype*& x)

makes x the information of node v.

void G.assign(*edge* e, *const etype*& x)

makes x the information of edge e.

node G.new_node(*const vtype*& x)

adds a new node $<x>$ to G and returns it.

node G.new_node(*node* u, *const vtype*& x, *int dir*)

adds a new node $v = <x>$ to G and returns it. v is inserted before ($dir = leda::before$) or after ($dir = leda::after$) node u into the list of all nodes.

edge G.new_edge(*node* v, *node* w, *const etype*& x)

adds a new edge $<v, w, x>$ to G by appending it to $adj\_edges(v)$ and to $in\_edges(w)$ and returns it.

edge G.new_edge(*edge* e, *node* w, *const etype*& x, *int* $dir = leda::after$)

adds a new edge $<source(e), w, x>$ to G by inserting it after ($dir = leda::after$) or before ($dir = leda::before$) edge e into $adj\_edges(source(e))$ and appending it to $in\_edges(w)$. Returns the new edge.