Reference Manual
for
Graph Data Structures

Vikas Kapoor *   Alexander Wolff *

Revision: 1.1, Date: 1998/01/22

*Institut für Informatik, Fachbereich Mathematik und Informatik, Freie Universität Berlin, Takustraße 9, 14195 Berlin, Germany, {kapoor | awolff}@inf.fu-berlin.de

Undirected Parameterized Graphs (*Ugraph<NodeType, EdgeType>*)

Definition

A *graph* $G(V, E)$ consists of a set V of nodes and a set E of edges. An edge is a pair of nodes. $G(V, E)$ is *undirected* iff for each pair $(v, w) \in E$ also $(w, v) \in E$. A *parameterized* undirected graph G is an undirected graph whose nodes and edges contain additional (user defined) data.

#include <ugraph.h>

Template Parameters

The class *Ugraph<NodeType, EdgeType>* is templated with the parameters *NodeType* and *EdgeType* which are the types of the data to be stored in the nodes and edges of the graph respectively.

Types

Ugraph<NodeType, EdgeType>:: pair<NodeType, EdgeType> value_type;

Type of the data stored in the graph. It is a pair of the data types stored in the nodes and the edges, respectively.

Ugraph<NodeType, EdgeType>:: Node_it

Type of the iterator that allows to traverse the nodes. The iterator is bidirectional.

Ugraph<NodeType, EdgeType>:: Edge_it

Type of the iterator that allows to traverse the edges. The iterator is bidirectional.

Ugraph<NodeType, EdgeType>:: Adj_node_it

Type of the iterator that allows to traverse the nodes adjacent to a given node. The iterator is bidirectional.

Ugraph<NodeType, EdgeType>:: Adj_edge_it

Type of the iterator that allows to traverse the edges incident to a given node. The iterator is bidirectional.

Creation

Ugraph<NodeType, EdgeType> ugraph;

creates the object *ugraph* of type *Ugraph<NodeType, EdgeType>*.

Modifiers

Node_it	*ugraph.new_node(NodeType node_inf)*	
		adds a node with data *node_inf* to *ugraph* and returns an iterator pointing to the new node.
Edge_it	*ugraph.new_edge(Node_it v, Node_it w, EdgeType edge_inf)*	
		adds an edge between v and w with data *edge_inf* to *ugraph* and returns an iterator pointing to the new edge.
void	*ugraph.del_node(Node_it n)*	deletes node n from *ugraph*.
void	*ugraph.del_edge(Edge_it e)*	deletes edge e from *ugraph*.
void	*ugraph.hide_edge(Edge_it e)*	hides edge e in *ugraph*.
void	*ugraph.restore_edge(Edge_it e)*	
		restores the hidden edge e in *ugraph*.

Access Functions for Nodes

int	*ugraph.num_of_nodes()*	returns the number of nodes in *ugraph*.
int	*ugraph.degree(Node_it n)*	returns the number of edges adjacent to node n.
Node_it	*ugraph.nodes_begin()*	returns an iterator referring to the first node in *ugraph*. This iterator can be used to traverse all nodes in *ugraph*.
Node_it	*ugraph.nodes_end()*	returns the corresponding past-the-end iterator.

Adj_node_it *ugraph.adj_nodes_begin(Node_it n)*

returns an iterator referring to the first node adjacent to node n. This iterator can be used to traverse all nodes adjacent to n.

Adj_node_it *ugraph.adj_nodes_end(Node_it n)*

returns the corresponding past-the-end iterator.

Node_it *ugraph.opposite(Node_it n, Edge_it e)*

returns an iterator pointing to the node adjacent to node n and incident to edge e.

Node_it *ugraph.get_node_it(Adj_node_it n)*

casts the given *Adj_node_it* to a *Node_it*.

Access Functions for Edges

int *ugraph.num_of_edges()* returns the number of edges in *ugraph*.

Edge_it *ugraph.edges_begin()* returns an iterator referring to the first edge in *ugraph*. This iterator can be used to traverse all edges in *ugraph*.

Edge_it *ugraph.edges_end()* returns the corresponding past-the-end iterator.

Adj_edge_it *ugraph.adj_edges_begin(Node_it n)*

returns an iterator referring to the first edge incident to node n. This iterator can be used to traverse all edges incident to n.

Adj_edge_it *ugraph.adj_edges_end(Node_it n)*

returns the corresponding past-the-end iterator.

Edge_it *ugraph.get_edge_it(Adj_edge_it n)*

casts the given *Adj_edge_it* to an *Edge_it*.

Implementation

We represent a graph by an STL list of STL lists of edges.

Destruction of a graph takes $\mathcal{O}(n+m)$ time, where n is the current number of nodes and m the current number of edges. All other operations take constant time.