Contents

6

Graphs and their Data Structures

The graph data type is one of the central data types in LEDA. In the first two sections we give a gentle introduction to it. Each of the remaining sections is devoted to a particular aspect of the graph data type: node and edge arrays, node and edge maps, node lists, node priority queues, node partitions, undirected graphs, graph generators, input and output, iteration statements, basic graph properties, parameterized graphs, and time and space complexity.

6.1 Getting Started

A *directed graph* $G = (V, E)$ consists of a set V of nodes or vertices and a set E of edges. Figure 6.1 shows a directed graph. Every edge *e* has a *source node source*(*e*) and a *target node target*(*e*). In our figures we draw an edge *e* as an arrow starting at *source*(*e*) and ending at *target*(*e*). We refer to the source and the target of an edge as the *endpoints* of the edge. An edge is said to be *incident* to its endpoints. We also say that an edge *e* is an edge *out of source*(*e*) and *into target*(*e*). The edges out of v are also called the edges *adjacent* to v. For an edge *e* with source node v and target node w we will write (v, w).

The declarations

```
graph G;
node  v, w;
edge  e, f;
```

declare variables G, v, w, e and f of type *graph*, *node*, and *edge*, respectively. The values of these variables are graphs, nodes, and edges, respectively; G is initialized to the empty graph, i.e., a graph with no node and no edge, and the initial values of v, w, e, and f are unspecified (since nodes and edges are pointer types). The special value *nil* is not a node or

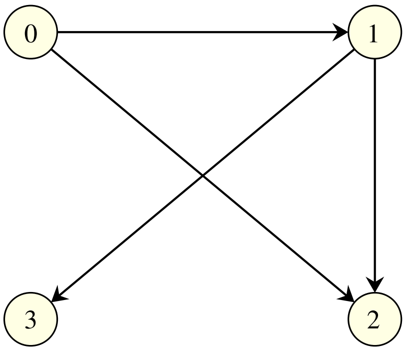

Figure 6.1 A directed graph.

edge of any graph and can be used to initialize nodes and edges with a definite value, as, for example, in

```
node v = nil;
```

Graph algorithms frequently need to iterate over the nodes and edges of a graph and the edges incident to a particular node. The iteration statement

```
forall_nodes(v,G){ }
```

iterates over all nodes of a graph, i.e., the nodes of G are successively assigned to v and the body of the loop is executed once for each value of v. Similarly,

```
forall_edges(e,G){ }
```

iterates over all edges e of G. There are three ways to iterate over the edges incident to a node v. The iteration statements

```
forall_out_edges(e,v){ }
forall_adj_edges(e,v){ }
```

iterate over all edges e out of v, i.e, all edges whose source node is equal to v,

```
forall_in_edges(e,v) { }
```

iterates over all edges e into v, i.e., over all edges whose target node is equal to v, and

```
forall_inout_edges(e,v){ }
```

iterates over all edges e into and out of v. So

```
int s = 0;
forall_edges(e,G) s++;
```

computes the number of edges of G. This number is also available as *G.number_of_edges*( ).

In many situations it is useful to associate additional information with the nodes and edges of a graph. LEDA offers several ways to do so. We briefly discuss *node arrays*, *edge arrays*, and *parameterized graphs*. We will give more details and also discuss node and edge maps later.

The declarations

```
node_array<string> name(G);
edge_array<int>    length(G,1);
```

introduce arrays *name* and *length* indexed by the nodes and edges of the G, respectively. The entries of *name* are strings and the entries of *length* are integers. All entries of *name* are initialized to the empty string (= the default value of *string*) and all entries of *length* are initialized to 1. If v is a vertex of G and e is an edge of G we may now write

```
name[v]   = "Saarbruecken";
length[e] = 5;
```

The following piece of code numbers the nodes of a graph with the integers 0 to $n-1$, where n is the number of nodes of G. As is customary in the literature on graph algorithms we will usually write n for the number of nodes and m for the number of edges.

```
node_array<int> number(G);
int count = 0;
forall_nodes(v,G) number[v] = count++;
```

A second method to associate information with nodes and edges is to use so-called *parameterized graphs*. The declaration

```
GRAPH<string,int> H;
```

declares H as a parameterized graph where a string variable is associated with every vertex of H and an integer is associated with every edge of H. We may now write

```
H[v] = "Saarbruecken";
H[e] = 5;
```

to associate the string "*Saarbruecken*" with v and the integer 5 with e. Of course, both operations are only legal if v and e actually denote a vertex and edge of H, respectively.

There is an important difference between the two methods of associating information with nodes and edges. Node and edge arrays work only for static graphs, i.e., when a new node or edge is added to a graph it will not have a corresponding entry in the node and edge arrays of the graph (in Section 6.3 this condition will be relaxed somewhat). Parameterized graphs, on the contrary, are fully dynamic. Information can be associated with new edges and nodes without any restriction. In this sense, parameterized graphs are more flexible. Also, the access to the information stored in the nodes and edges of a parameterized graph is somewhat more efficient than the access to the information stored in a node or edge array. On the other hand, the great strength of node and edge arrays is that an arbitrary number of them can be defined for a graph.

It's time to learn how to build non-trivial graphs. A graph can be altered by adding and deleting nodes and edges. For example,

```
graph G;
G.new_node();
G.new_node();
node v;
forall_nodes(v,G) cout << G.outdeg(v);
```

makes G a graph with two nodes and no edge and then outputs the outdegree[1] of all nodes, i.e., outputs the number 0 twice. In order to add an edge we need to specify its source and its target. For example,

```
node w = G.first_node();
G.new_edge(w,G.succ_node(w));
```

will add an edge whose source and target are the first and second node of G respectively; note that LEDA internally orders the nodes of a graph in the order in which they were added to G. *G.first_node*( ) returns the first node in this ordering and *G.succ_node*(w) returns the node added immediately after w. There is a more interesting way to add edges. The operation *G.new_node*( ) does not only add a new node to the graph G but also returns the new node. We can remember the new node in a variable of type *node*. So

```
graph G;
node v0 = G.new_node();
node v1 = G.new_node();
node v2 = G.new_node();
node v3 = G.new_node();
G.new_edge(v0,v1); G.new_edge(v0,v2);
G.new_edge(v1,v2); G.new_edge(v1,v3);
```

creates the graph of Figure 6.1.

Let us do something more ambitious next. Suppose that we created a graph G and that we want to make an isomorphic copy H of it. Moreover, we want every node and edge of H to know its original in G. Here is an elegant way to do this. We use parameterized graphs, node arrays and edge arrays.

```
void CopyGraph(GRAPH<node,edge>& H, const graph& G)
{ H.clear();  // reset H to the empty graph
  node_array<node> copy_in_H(G);
  node v;
  forall_nodes(v,G) copy_in_H[v] = H.new_node(v);
  edge e;
  forall_edges(e,G)
    H.new_edge(copy_in_H[source(e)],copy_in_H[target(e)],e);
}
```

We define H as a parameterized graph where a node can be associated with each node and an edge can be associated with each edge. We also define a node array *copy_in_H* for G that allows us to associate a node with every node of G. We then iterate over the nodes of G. For every node v of G the operation H.*new_node*(v) adds a new node to H and associates v with the new node. Note that the *new_node* operation for a parameterized graph has an argument, namely the information that is to be associated with the new node. The operation H.*new_node*(v) also returns the new node. We remember it in *copy_in_H*[v]. The overall effect of the *forall_nodes*-loop is to give H as many nodes as G and to establish bidirectional

[1] The outdegree of a vertex v is the number of edges e with *source*(e) $= v$.

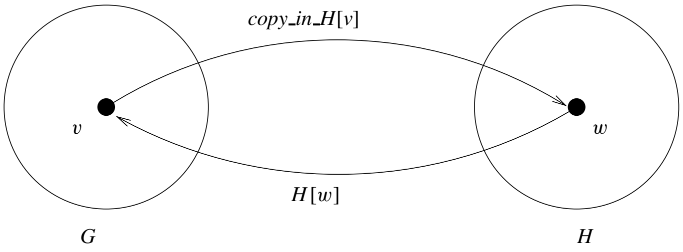

Figure 6.2 A graph G and an isomorphic copy H of it. Each node v of G knows its partner in H through $copy\_in\_H[v]$ and each node w of H knows its partner in G through $H[w]$.

links between the nodes of G and H: in particular, we have $H[copy\_in\_H[v]] = v$ for all nodes v of G and $copy\_in\_H[H[w]] = w$ for all nodes w of H, see Figure 6.2. It is now easy to add the edges. We iterate over the edges of G. For every edge e we add an edge to H that runs from $copy\_in\_H[source(e)]$ to $copy\_in\_H[target(e)]$ and also make e the information associated with the new edge. Observe that $H.new\_edge(x, y, inf)$ adds an edge from node x to node y and associates the information inf with it.

Exercise for 6.1

1 Write a program that makes a copy of a graph G with all edges reversed, i.e., for every edge $e = (v, w)$ in G there should be an edge from the copy of w to the copy of v in H.

6.2 A First Example of a Graph Algorithm: Topological Ordering

A graph is called *acyclic* if it contains no cycle. A cycle is a path that closes on itself, i.e., a sequence $e_0, e_1, \ldots, e_k$ of edges such that $target(e_i) = source(e_{i+1 \bmod k+1})$ for all i, $0 \le i \le k$. The graph in Figure 6.1 is acyclic. The nodes of an acyclic graph can be numbered such that all edges run from smaller to higher numbered nodes. The function

```
bool TOPSORT(const graph& G, node_array<int>& ord);
```

returns true if G is acyclic and false if G contains a cycle. In the former case it also returns a topological ordering of the nodes of G in *ord*.

The procedure works by repeatedly removing nodes of indegree zero and numbering the nodes in the order of their removal.

In the example of Figure 6.1 we first number node 0. Removing node 0 makes the indegree of node 1 zero and hence this node is numbered next. Removal of node 1 makes the indegree of node 2 zero, ...

For reasons of efficiency we keep track of the current indegree of all nodes and also maintain the list of nodes whose current indegree is zero.

```
#include <LEDA/graph.h>
#include <LEDA/queue.h>
bool procedure TOPSORT(const graph& G,node_array<int>& ord)
{ ⟨initialization⟩
  ⟨removing nodes of indegree zero⟩
}
```

In the initialization phase we determine the indegree of all nodes and initialize a queue of nodes of indegree zero.

⟨*initialization*⟩≡

```
node_array<int> INDEG(G);
queue<node>     ZEROINDEG;
node v,w;
forall_nodes(v,G)
  if ( (INDEG[v] = G.indeg(v)) == 0 ) ZEROINDEG.append(v);
```

In the main phase of the algorithm we consider the nodes of indegree zero in turn. When a vertex v is considered we number it and we decrease the indegrees of all adjacent nodes by one. Nodes whose indegree becomes zero are added to the rear of *ZEROINDEG*.

⟨*removing nodes of indegree zero*⟩≡

```
int count = 0;
node_array<int> node_ord(G);
while (!ZEROINDEG.empty())
{
  v = ZEROINDEG.pop();
  node_ord[v] = ++count;
  forall_out_edges(e,v)
  { node w = G.target(e);
    if ( --INDEG[w] == 0 ) ZEROINDEG.append(w);
  }
}
return (count == G.number_of_nodes());
```

TOPSORT considers every edge of G only once and hence has running time $O(n + m)$. In the section on depth-first search (see Section 7.3) we will see an alternative program for topological sorting.

6.3 Node and Edge Arrays and Matrices

Node and edge arrays and matrices are the main means of associating information with the nodes and edges of a graph. The declarations

```
node_array<E> A(G);
node_array<E> B(G,E x);
```

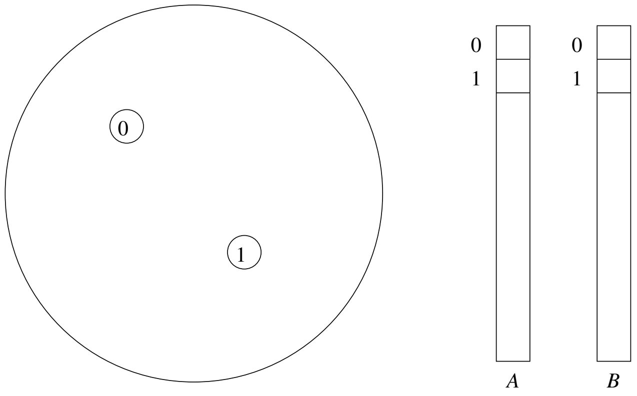

Figure 6.3 The realization of node arrays: Node arrays A and B are realized by regular arrays. The nodes of a graph are numbered and the node numbers are used as the indices into the arrays.

declare node arrays A and B for the nodes of G, respectively. The elements of A are initialized with the default value of E and the elements of B are initialized to x. Edge arrays are declared in a similar way. So

```
node_array<bool> visited(G,false);
```

declares a node array *visited* and initializes all its entries to false. The cost of declaring a node array for G is proportional to the number of nodes of G and the cost of declaring an edge array is proportional to the number of edges.

Node and edge arrays are a very flexible way of associating information with the nodes and edges of a graph: any number of node or edge arrays can be defined for a graph and they can be defined at any time during execution.

Node and edge arrays are implemented as follows. The nodes and edges of a graph are numbered in the order of their construction, starting at zero. We call the number of a node or edge its *index*. The index of a node v or edge e is available as $\mathit{index}(v)$ and $\mathit{index}(e)$, respectively. Node and edge arrays are realized by standard arrays. The node and edge indices are used to index into the arrays, see Figure 6.3.

The access to an entry $A[v]$ of a node array A (similarly, edge arrays) requires two accesses to memory, first the structure representing the node v is accessed to determine $\mathit{index}(v)$ and second the entry $A[\mathit{index}(v)]$ is accessed.

When the number of node and edge arrays that are needed for a graph is known, the following alternative is possible. Assume that *n_slots* node arrays and *e_slots* edge arrays are needed. The constructor

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

constructs a graph where the structures representing nodes have room for the entries of *n_slots* node arrays and the structures representing edges have room for the entries of *e_slots* edge arrays. In order to use one of the slots for a particular array, one writes:

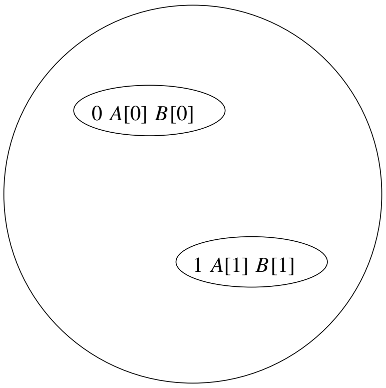

Figure 6.4 The alternative realization of node arrays: In a graph G constructed by *graph* $G(2, 0)$, every node has room for the entries of two node arrays.

```
node_array<E> A;
A.use_node_data(G, E x);
```

This will reserve one of the slots in the node structures for A and initialize all entries of the array to x. If no slot is available, the node array is realized by a standard array. Figure 6.4 illustrates the alternative. The alternative realization of node and edge arrays is frequently, but not always, faster (see the next section), as only one access to memory is needed to access an entry of a node or edge array, but it is also less convenient, as the number of node and edge arrays that can use the alternative is fixed at the time of the construction of the graph.

We recommend that you experiment with the alternative design during the optimization phase of program development.

Node and edge arrays, as discussed so far, are primarily useful for static graphs.

```
node_array<int> dist(G);
node v = G.new_node();
dist[v] = 5;
```

is illegal and produces the error message "*node_array*[v] not defined for v". We next discuss node and edge arrays for dynamic graphs. We have to admit, though, that we hardly use node and edge arrays for dynamic graphs ourselves. We prefer node and edge maps and parameterized graphs.

```
node_array<E> A(graph G, int n, E x);
```

declares A as a node array of size n for the nodes of G and initializes all entries of A to x; x must be specified even if it is the default value of E.

The constructor requires that $n \geq |V|$. The array A has room for $n - |V|$ additional nodes, i.e., for the nodes created by the next $n - |V|$ calls of *G.new_node*( ). In this way one can have the convenience and efficiency of node arrays also for dynamic graphs. Deletion of nodes is no problem for node arrays.

The following doubling and halving strategy is useful for node and edge arrays on dynamic graphs. Suppose that n_0 is the current number of nodes of G and that we want to create a node array A for G. We make A an array of size $2n_0$ and initialize two counters *ins_count* and *del_count* to zero. We increment *ins_count* for every *G.new_node*( ) operation and *del_count* for every *G.del_node*( ) operation. When *ins_count* reaches n_0 or *del_node* reaches $n_0/2$ we allocate a new array B of size $2(n_0 + \mathit{ins\_count} - \mathit{del\_count})$ and move the contents of A to B. This scheme ensures that node arrays are always at least 25% utilized and that the overhead for moving information around increases the running time by only a constant factor (since the cost of moving is $O(n_0)$ and since there are $\Omega(n_0)$ *new_node* and *del_node* operations between reorganizations of the node array).

We next turn to node matrices. The definition

```
node_matrix<int> M(G,0);
```

defines M as a two-dimensional matrix indexed by pairs of nodes of G and initializes all entries of G to zero. This takes time $O(n^2)$, where n is the number of nodes of G. The space requirement for a node matrix is quadratic in the number of nodes. So they should only be used for small graphs.

```
M(v,w) = 1;
```

sets the entry for pair (v, w) to one.

A node matrix can also be viewed as a node array of node arrays, i.e., the type *node_matrix*<E> is equivalent to the type *node_array*<*node_array*<E> >. This view is reflected in the operation

```
M[v];
```

which returns a node array.

We give an example of the use of node matrices. The following three-liner checks whether a graph is bidirected (also called symmetric), i.e., whether for every edge $e = (v, w)$ the reversed edge (w, v) is also present.

```
node_matrix<bool> M(G,false);
forall_edges(e,G) M(G.source(e),G.target(e)) = true;
forall_edges(e,G)
{ if ( !M(G.target(e),G.source(e)) ) error_handler(1,"not bidirected"); }
```

The program above has running time $\Theta(n^2 + m)$, $\Theta(n^2)$ for initializing M and $\Theta(m)$ for iterating over all edges twice. As we will see later there is also an $O(m)$ algorithm for the same task. It is available as

```
bool Is_Bidirected(G);
```

6.4 Node and Edge Maps

Nodes and edge maps are an alternative to node arrays. The declarations

```
node_map<E> A(G);
node_map<E> B(G,E x);
```

declare node maps A and B for the nodes of G, respectively. The elements of A are initialized with the default value of E and the elements of B are initialized to x. Edge maps are declared in a similar way. So

```
node_map<bool> visited(G,false);
```

declares a node map *visited* and initializes all its entries to false.

What is the difference between node and edge arrays and node and edge maps? Node and edge maps use hashing (see Section 5.1.2). The declaration of a node or edge map has constant cost (compare this to the linear cost for node and edge arrays) and the access to an entry of a node or edge map has constant expected cost.

Table 6.1 compares three ways of associating information with the nodes of a graph, the standard version of node arrays, the version of node arrays that makes use of a data slot in the node, and node maps. The table was produced by the program below. We give the complete program because the numbers in the table are somewhat surprising. We create a graph with n nodes and no edge and iterate R times over the nodes of the graph. In each iteration we access the information associated with the node. We iterate over the nodes once in their natural order and once in random order.

⟨*node_arrays_versus_node_maps*⟩≡

```
main(){
⟨node arrays versus node maps: read n and R⟩
graph G; graph G1(1,0); node v; int j;
random_graph(G,n,0); random_graph(G1,n,0);
float T = used_time();
float TA, TB, TM, TAP, TBP, TMP;
{ node_array<int> A(G,0);
  for ( j = 0; j < R; j++ )
    forall_nodes(v,G) A[v]++;
  TA = used_time(T);
}

{ node_array<int> A;
  A.use_node_data(G1,0);
  for ( j = 0; j < R; j++ )
    forall_nodes(v,G1) A[v]++;
  TB = used_time(T);
}
{ node_map<int> A(G,0);
  for ( j = 0; j < R; j++ )
    forall_nodes(v,G) A[v]++;
  TM = used_time(T);
```

Linear scan			Random scan		
array	node data	map	array	node data	map
3.25	4.39	3.48	8.96	5.9	9.56

Table 6.1 Node arrays versus node maps: The table shows the output of the program node_arrays_versus_node_maps.c. We used a node array (columns one and four), a node data slot (columns two and five), and a node map (columns three and six). We used a graph with one million nodes and $R = 10$. The nodes were scanned in linear order and in random order. The node_array_versus_node_maps demo allows you to perform your own experiments.

```
}

array<node> perm(n); array<node> perm1(n);
int i = 0;
forall_nodes(v,G) perm[i++] = v;
i = 0;
forall_nodes(v,G1) perm1[i++] = v;
perm.permute(); perm1.permute();
used_time(T);
{ node_array<int> A(G,0);
  for ( j = 0; j < R; j++ )
    for(i = 0; i < n; i++) A[perm[i]]++;
  TAP = used_time(T);
}
{ node_array<int> A;
  A.use_node_data(G1,0);
  for ( j = 0; j < R; j++ )
    for(i = 0; i < n; i++) A[perm1[i]]++;
  TBP = used_time(T);
}
{ node_map<int> A(G,0);
  for ( j = 0; j < R; j++ )
    for(i = 0; i < n; i++) A[perm[i]]++;
  TMP = used_time(T);
}
⟨node arrays versus node maps: report running times⟩
}
```

In the random scan over the nodes, node data slots outperform node arrays which in turn outperform node maps. This was to be expected, since node data slots avoid one level of indirection, and since maps have the overhead of hashing. Maps are only slightly slower than arrays due to our very efficient realization of maps, see Section 5.1.2. In the linear scan the situation is different. Node data slots are the slowest and maps are even closer to arrays. We believe that this is due to caching. We compare node arrays and node data slots. When node data slots are used, the node structures are larger, and hence fewer of them fit into a cache line. Node arrays use the cache more effectively in the linear scan because they can use one cache line for node structures and one cache line for the array itself and only the cache lines for the array itself are written. Thus the number of write-faults reduces. A similar explanation applies to node maps. Since it requires knowledge of the implementation of maps, we do not give it here.

We recommend to use node and edge maps in situations where a sparse map on nodes or edges, respectively, has to be maintained. If more than half of the entries are actually used, it is better to use node arrays.

We next turn to two-dimensional node maps. The definition

```
node_map2<int> M(G,0);
```

defines M as a two-dimensional map indexed by pairs of nodes of G and initializes all entries of G to zero. This takes constant time.

```
M(v,w) = 1;
```

sets the entry for pair (v, w) to one. The space requirement for a two-dimensional node map is proportional to the number of entries used.

We give an example for the use of two-dimensional node maps. The following three-liner checks whether a graph is bidirected (also called symmetric), i.e., whether for every edge $e = (v, w)$ the reversed edge (w, v) is also present.

```
node_map2<bool> M(G,false);
forall_edges(e,G) M(G.source(e),G.target(e)) = true;
forall_edges(e,G)
{ if ( !M(G.target(e),G.source(e)) ) error_handler(1,"not symmetric"); }
```

The program above has running time $O(m)$, $O(1)$ for initializing M and $O(m)$ for iterating over all edges twice. The space requirement is $O(m)$. Observe, that this is much better than what we obtained with node arrays in the preceding section if $m \ll n^2$.

Exercises for 6.4

1 Write a program that checks whether a graph is symmetric and, if so, computes an edge array *reversal* that stores for each edge a reversal of the edge. The source of *reversal*$[e]$ must be equal to the target of e and vice versa.

2 Extend the program of the previous item so that it can also handle parallel edges. We want *reversal*[*reversal*$[e]] = e$ for all edges e.

3 Extend the program of the previous item so that it can also handle self-loops. We want *reversal*$[e] \neq e$ for all e.

6.5 Node Lists

A node list is a combination of a doubly linked list of nodes and a node map which gives, for each node, its position in the list, see Figure 6.5. *A node can be contained in a node list*

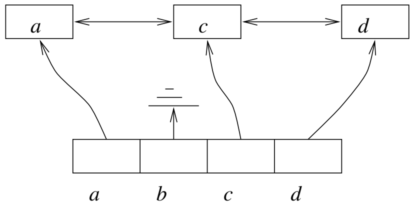

Figure 6.5 Node lists: A node list for a graph with four nodes a, b, c, and d. The node list contains the nodes a, c, and d in this order. The top part of the figure shows a doubly linked list and the lower part of the figure indicates a node map. The node map maps each node contained in the node list to the list item containing the node.
In a *snode_list* a singly linked list is used instead of a doubly linked list.

at most once. It can be contained in several node lists, but in each particular node list it can appear only once.

```
node_list L(G);
```

creates a node list for the graph G and initializes it with the empty list. Node lists offer all the usual list operations, e.g., *append*, *push*, *pop*, *insert*, *head*, *tail*, *pred*, *succ*, *cyclic_pred*, *cyclic_succ*, *empty*, and the possibility to iterate over the nodes in the list. In addition, node lists offer constant time member ship test.

The related data type *snode_list* is the combination of a singly linked list and a node map. It offers all the operations of singly linked lists plus constant time member ship test.

A prime example for the use of node lists is breadth-first search. The goal is to explore the nodes of a graph starting from some source node s in order of increasing distance from s. The distance of a node v from s is the smallest number of edges in a path from s to v.

The following program realizes breadth-first search. We collect the nodes of G in a *snode_list* Q in the order in which they are reached. We always explore the edges out of the first unprocessed node in Q. Whenever a node is encountered that has not been reached before (= is not in Q) we add it to the rear of Q.

```
snode_list Q;
Q.append(s);
node v = Q.head();
while ( v != nil )
{ edge e;
  forall_adj_edges(e,v)
  { node w = G.target(e);
    if ( !Q.member(w) ) Q.append(w);
  }
  v = Q.succ(v);
}
```

We will discuss breadth-first search in more detail in the chapter on graph algorithms.

Exercises for 6.5

1 Give an implementation of *snode_list* that uses a *node_map*<*node*> *succ_node*, two nodes *first_node* and *last_node*, and an integer *size*.

2 Give an implementation of *node_list* that uses two maps from nodes to nodes, namely, *succ_node* and *pred_node*, two nodes *first_node* and *last_node*, and an integer *size*.

6.6 Node Priority Queues and Shortest Paths

The declaration

```
node_pq<P> Q(G);
```

declares a *node priority queue* Q with priority type P for G and initializes it to the empty queue. A node priority queue with priority type P is a partial function from the nodes of G to the set P. The set P must be linearly ordered. If $Q(v)$ is defined we call it the priority of node v. We use dom Q to denote the set of nodes for which $Q(v)$ is defined, the *domain* of Q. Node priority queues allow us to manipulate the function Q by insertion, deletion, and (restricted) modification of values, and they allow us to select a node with smallest priority.

We next discuss some of the operations available on node priority queues in more detail, then show how to use them in an implementation of Dijkstra's algorithm for the single-source shortest-path problem, and finally show how node priority queues are implemented in terms of node arrays and general priority queues.

We come to the operations available on node priority queues:

```
node Q.find_min();
```

returns a node $v \in \text{dom } Q$ with minimal associated priority (*nil* if Q is empty),

```
bool Q.member(node v);
```

checks whether node v is contained in the queue Q, i.e., if $v \in \text{dom } Q$,

```
void Q.insert(node v, P p);
```

adds the node v with associated priority p to the queue Q (the effect of this operation is unspecified if v is already contained in Q) and

```
void Q.decrease_p(node v, P p);
```

makes p the new priority of node v (the effect of this operation is unspecified if v is not contained in Q or p is larger than the old priority associated with v).

The implementation of node priority queues is based on priority queues and node arrays. The operations *find_min* and *decrease_p* take constant time, all other operations take time $O(\log s)$ where s is the current size of Q. The space requirement is proportional to the number of nodes of G. We give the details of the implementation at the end of the section.

We illustrate the use of node priority queues on Dijkstra's single-source shortest-path algorithm. Let G be a graph, let *edge_array<NT> cost* be a non-negative cost function[2] on the edges of G, and let s be a node of G. For any node v of G let $\mu(v)$ be the cost of a shortest path from s to v, where the cost (or length) of a path is the sum of the costs of its edges; if there is no path from s to v then $\mu(v) = \infty$. We use $cost(p)$ to denote the cost of a path p.

The task is to compute μ in a *node_array<NT> dist* and a *node_array<edge> pred* which contains for each node $v \neq s$ the last edge of a shortest path from s to v. We need to be more precise. Observe that not every number type has a representation for ∞, and hence the previous sentence does not specify how the algorithm should report the fact that $\mu(v) = \infty$ for a node v. We refine the specification to the following:

- If v is reachable from s then $dist[v] = \mu(v)$.
- $pred[s] = nil$.
- If $v \neq s$ and v is reachable from s then $pred[v]$ is the last edge of a shortest path from s to v.
- If $v \neq s$ and v is not reachable from s then $pred[v] = nil$.

Dijkstra's algorithm [Dij59] "simulates" the following physical process. Imagine the graph as a network of uni-directional wires, imagine that current is injected into the network at node s and time zero, and imagine that current spreads with unit speed. Thus current requires $cost[e]$ time units to spread across an edge e. In this model, the current will reach every node v at time $\mu(v)$.

In order to carry out the simulation, we turn the nodes of the network into active components. As soon as current reaches a node u, say at time $t = \mu(u)$, the node sends a message to each node v with $e = (u, v) \in E$ with the content:

You will receive current through edge e at time $t + cost[e]$.

Every node v keeps track of all the messages sent to it. More precisely, a node keeps track of the earliest time at which current will reach it, i.e., whenever a node v receives a message, it checks whether the message promises it an earlier delivery time and, if so, the node updates its time estimate. In our implementation we keep the current time estimate of node v in $dist[v]$ and we keep the edge through which the node will receive current at time $dist[v]$ in $pred[v]$. If v has received no message yet we have $pred[v] = nil$.

The simulation is driven by a global clock which we call wall time. At any time t there will be a set S of nodes which have already been reached by the current and which have accordingly sent messages to their neighbors, and there will be the set $V \setminus S$ of the remaining nodes which have not been reached yet by the current wave. Each node in $V \setminus S$ has received zero or more messages and keeps track of its earliest delivery time. Clearly, the node which

[2] *NT* denotes an arbitrary number type.

is reached next by the current is the node $u \in V \setminus S$ with the smallest delivery time, i.e., the smallest value $dist[u]$. It is the next node to send out messages.

In an implementation the crucial question is how to find the node v with minimal *dist*-value among the nodes in $V \setminus S$. The data type node priority queue is ideally suited for that purpose. Simply have a *node_pq*<*NT*> P with

$$\text{dom}\, P = \{v \,;\, v \in V \setminus S \text{ and } pred[v] \neq nil\}$$

and $P(v) = dist[v]$ for any $v \in \text{dom}\, P$, i.e., P contains all nodes outside S which have received at least one message and records, for each such node, the earliest delivery time to the node. Then $P.del\_min()$ returns the desired node and deletes it from P. The complete program follows.

⟨*dijkstra.t*⟩+≡

```
template <class NT>
void DIJKSTRA_T(const graph& G, node s, const edge_array<NT>& cost,
                   node_array<NT>& dist, node_array<edge>& pred)
{
  node_pq<NT>  PQ(G);
  node v; edge e;
  dist[s] = 0;
  PQ.insert(s,0);
  forall_nodes(v,G) pred[v] = nil;
  while (!PQ.empty())
  { node u = PQ.del_min(); // add u to S
    NT du = dist[u];
    forall_adj_edges(e,u)
    { v = G.opposite(u,e); // makes it work for ugraphs
      NT c = du + cost[e];
      if (pred[v] == nil && v != s )
        PQ.insert(v,c); // first message to v
      else if (c < dist[v]) PQ.decrease_p(v,c); // better path
           else continue;
      dist[v] = c;
      pred[v] = e;
    }
  }
}
```

The program runs in time $O(m + n \log n)$ since every node is deleted from the queue at most once and *del_min* has cost $O(\log n)$ and since every other operation is executed at most $O(n + m)$ times and has constant amortized cost.

In the remainder of this section we show how to implement node priority queues in terms of node arrays and priority queues. The construction is very simple. We realize a *node_prio*<P> *NPQ* for a graph G by a *p_queue*<P, *node*> *PQ* and a *node_array*<*pq_item*> *item_of* such that:

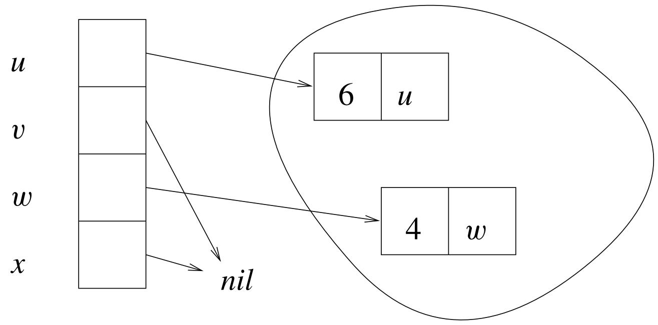

Figure 6.6 A node priority queue for a graph with four nodes u, v, w, and x. The priority of u is 6, the priority of w is 4, and v and x have no entry in the queue.

- if a node v is stored in *NPQ* with priority p then there is an item $pit = \langle p, v\rangle$ in *PQ* and $item\_of[v] = pit$.
- if a node v is not contained in *NPQ* then $item\_of[v] = nil$.

Figure 6.6 illustrates these invariants and node_pq.c shows the complete implementation.

$\langle$*node_pq.c*$\rangle\equiv$

```
#include <LEDA/graph.h>
#include <LEDA/p_queue.h>

template <class P> class node_pq {
private:
  p_queue<P,node> PQ;
  node_array<pq_item> item_of;

public:
  node_pq(const graph& G): item_of(G,nil) { }
  ~node_pq() { }

  void insert(node v, P p)     { item_of[v]= PQ.insert(p,v); }

  P prio(node v)               { return PQ.prio(item_of[v]); }

  void decrease_p(node v, P p) { PQ.decrease_p(item_of[v],p); }

  void del(node v)
  { PQ.del_item(item_of[v]);
    item_of[v] = nil;
  }

  node find_min()              { return PQ.inf(PQ.find_min()); }

  node del_min()
  { node v= PQ.inf(PQ.find_min());
    PQ.del_min();
    item_of[v] = nil;
    return v;
  }

  ⟨node_pq::other operations⟩
};
```

Only a few words are required to explain this code. We construct a *node_pq*<*P*> for a graph G by constructing an empty priority queue *PQ* and a node array *item_of* for G and by initializing all entries of *item_of* to *nil*. The former is done by the default constructor of priority queues and requires no code and the latter is achieved by the constructor call *item_of*(G, nil). In order to insert a pair (v, p) we insert the pair (p, v) into *PQ* and store the item that is returned in *item_of*$[v]$. In order to look up the priority of a node v we return *PQ.prio*(*item_of*$[v]$), … .

Exercises for 6.6

1 Modify Dijkstra's algorithm such that it does not start with a single source node s but with a set L of sources. It is supposed to compute $\mu(L, v)$ for all nodes v where $\mu(L, v)$ is the minimum distance from a node in L to v.

2 (Single sink shortest path). Let s and t be distinct nodes in a directed graph with non-negative edge weights. The goal is to compute a shortest path from s to t. Assume that there is heuristic information available which gives for any node v a *lower bound* $lb(v)$ for the length of a shortest path from v to t. Modify Dijkstra's algorithm such that $dist(v) + lb(v)$ is used as the priority of node v.

3 Use the algorithm of the previous item to compute shortest paths in graphs embedded into the plane, e.g., Delaunay diagrams (see Section 10.4). Define the cost of an edge as the Euclidean distance between its endpoints and let $lb(v)$ for any node v be the Euclidean distance between v and t. Which improvement in running time results from the use of heuristic information?

4 Implement operations *member*, *clear*, *size*, and *empty* of *node_pq*.

6.7 Undirected Graphs

In an *undirected* graph the edges have no direction. Mathematically speaking, an edge in an undirected graph is an unordered pair $\{v, w\}$ of nodes and an edge in a directed graph is an ordered pair (v, w) of nodes. As for directed graphs, we call v and w the endpoints of the edge. The endpoints of an edge in an undirected graph must be distinct (since an edge is a set of vertices of cardinality two).

6.7.1 *Viewing Directed Graphs as Undirected Graphs*

Every directed graph without self-loops can be viewed as an undirected graph.

For an edge e and an endpoint v of e

```
G.opposite(v,e)
```

returns the other endpoint of e, i.e., returns $target(e)$ if $v = source(e)$ and returns $source(e)$ otherwise.

The iteration statement

```
forall_inout_edges(e,v){  }
```

iterates over all edges e having v as one of their endpoints. It iterates first over all edges out of v and then over all edges into v.

The iteration *forall_inout_edges* and the function *opposite* can also be applied to graphs with self-loops. Observe, however, that the iteration statement will consider a self-loop $e = (v, v)$ twice, once as an edge, whose source is equal to v, and once as an edge, whose target is equal to v.

It is our experience that the two statements above suffice to deal with undirected graphs. We can foresee one situation where they do not suffice: if one wants to iterate over the edges incident to v in some mixed order, first some edges out of v, then some edges into v, then again some edges out of $v, \ldots$. We will see in Section 6.11 that the order of the out-edges and the order of the in-edges can be modified. Nevertheless, out-edges always come before in-edges in the *forall_inout_edges* iteration. If a more flexible scanning order is required, the following operation is useful:

```
G.make_undirected();
```

appends for every node v the list of in-edges of v to the list of out-edges of v and removes all self-loops. All edges incident to any node are now in a single list and hence can be rearranged freely using the operations to be described in Section 6.11.

```
G.make_directed();
```

partially reverses the operation above. It moves, for every node v, all edges e with $target(e) = v$ from the list of out-edges of v to the list of in-edges of v. Note that the operation does not reinsert self-loops.

6.7.2 *The Data Type ugraph*

We also have a data type *ugraph*. We use it very rarely. Ugraphs offer the same operations as graphs but the *new_edge* operation is interpreted differently. For example,

```
ugraph G;
node v = G.new_node(); node w = G.new_node();
edge e = G.new_edge(v,w);
```

creates an undirected graph with two nodes and one edge. The edge e is inserted into the out-lists of v and w (which in this context is better called the list of adjacent edges). Thus

```
e == G.first_adj_edge(v) && e == G.first_adj_edge(w)
```

evaluates to true. As for directed graphs the functions *source*( ) and *target*( ) yield the two endpoints of an edge, so *G.source*(e) returns v and *G.target*(e) returns w. Note that the role of the two nodes v and w in the definition of the edge e is not symmetric: v is made the source of e because it is mentioned first, and w is made the target of e because it is mentioned second.

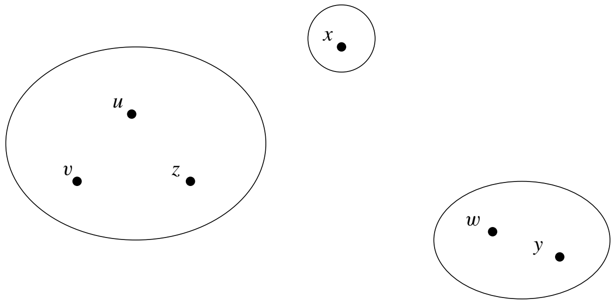

Figure 6.7 A node partition for a graph with six nodes u, v, w, x, y, and z: u, v, and z are in the same block, w and y are in the same block and x is a block of its own.

6.8 Node Partitions and Minimum Spanning Trees

We discuss node partitions. We first discuss their functionality and then illustrate their use in Kruskal's minimum spanning tree algorithm.

A *node partition* is a partition of the nodes of a graph G, i.e., a family of pairwise disjoint sets (called *blocks*) whose union is the set of nodes of G, see Figure 6.7 for an example.

```
node_partition P(G);
```

declares P as a node partition for G and initializes it to the finest partition of G, i.e., every node v of G forms its own block $\{v\}$. Node partitions offer the following operations:

```
bool P.same_block(node v,node w);
```

returns *true* iff v and w belong to the same block of P,

```
void P.union_blocks(node v,node w);
```

combines the blocks containing v and w. Each block has a *canonical representative*. The canonical representative of a block is some element in the block; it is not specified which. The operations

```
node P.find(node v);
node P(node v);
```

return the canonical representative of the block containing v. So, in the example of Figure 6.7, *P.find*(x) returns x (for the singleton block there is no choice of canonical element) and *P.find*(u) and *P.find*(v) return the same element of block $\{u, v, z\}$ (it is not specified which). When the functional notation $P(v)$ is used for the find operation it is convenient to name the partition after the name for the canonical element; for example, in the matching algorithm of Section 7.7 we will call the node partition *base*. After a union operation the data structure chooses the canonical representative of the block formed (among the elements of the block). We can make v the canonical representative of the block containing v by

```
void P.make_rep(node v);
```

The operation

```
void P.split(list<node> T);
```

splits the blocks containing the nodes in T into singleton blocks. The operation requires that T is a union of blocks of P. So, in the example of Figure 6.7 we can apply split with the argument $\{u, v, z, x\}$ but not with the argument $\{u, y\}$ and not with the argument $\{u, v\}$.

The implementation of node partitions is based on the data types *partition* and *node_array*. A sequence of m operations (except for split) on a node partition of n nodes takes time $O((n+m)\alpha(n))$ where α is the functional inverse of the Ackermann function. The function α is extremely slowly growing, in particular $\alpha(n) \leq 5$ for $n \leq 10^{100}$. The running time of node partitions is therefore linear for all practical purposes. A split takes time proportional to the size of T.

We turn to Kruskal's minimum spanning tree algorithm.

Let G be a graph whose edges have an associated cost of some number type and let *cmp* be a function that compares edges according to their cost, i.e., *cmp*(*e1*, *e2*) returns -1, 0, and $+1$, respectively, if the cost of *e1* is smaller than, equal to, or larger than the cost of *e2*. A subset T of the edges of G is called a *spanning forest* of G if any two nodes that are connected in G are also connected using only edges in T and if the subgraph (V, T) is acyclic. A spanning forest of a connected graph is a tree. The cost of a spanning forest is the sum of the costs of its edges. A *minimum spanning forest* is a spanning forest of minimal cost, see Figure 6.8 for an example. Kruskal [Kru56] discovered a very simple method for computing minimum cost spanning forests; it is customary to refer to his algorithm as a spanning tree algorithm although it will not compute a tree on a graph consisting of more than one connected component.

Kruskal's algorithm starts with an empty set T of edges and considers the edges of G in order of increasing cost. When considering an edge $e = \{u, v\}$ it checks whether addition of e to T would close a cycle. If it does not close a cycle then e is added to T and if it closes a cycle then e is discarded. In this way, T gradually evolves into a minimum spanning forest.

We give a proof. Less mathematically inclined readers may skip the proof. For the following argument let $e_1, e_2, \ldots, e_m$ be the sequence of edges of G ordered in order of increasing cost and let F_0 be the lexicographically smallest minimum spanning forest[3]. We show that $T \cap \{e_1, \ldots, e_i\} = F_0 \cap \{e_1, \ldots, e_i\}$ for all i, $0 \leq i \leq m$, by induction on i. This is clearly true for $i = 0$. Consider $i > 0$. If e_i closes a cycle with respect to $T \cap \{e_1, \ldots, e_i\}$ then it closes a cycle with respect to F_0 and hence e_i belongs to neither of the two sets. If e_i does not close a cycle with respect to $T \cap \{e_1, \ldots, e_i\}$ then it is added to T. We need to show $e_i \in F_0$. Since F_0 is a spanning forest there must be a path p in F_0 connecting the endpoints of e_i and since the endpoints of e_i are not connected by the edges in $T \cap \{e_1, \ldots, e_{i-1}\} = F_0 \cap \{e_1, \ldots, e_{i-1}\}$ there must be an edge e_j with $j \geq i$ in

[3] We may view a spanning forest as a string over $\{0, 1\}$ of length m where a 1 in the i-th position indicates that e_i belongs to the spanning forest and a 0 indicates that it does not. The lexicographic ordering on these strings defines an ordering on spanning forests.

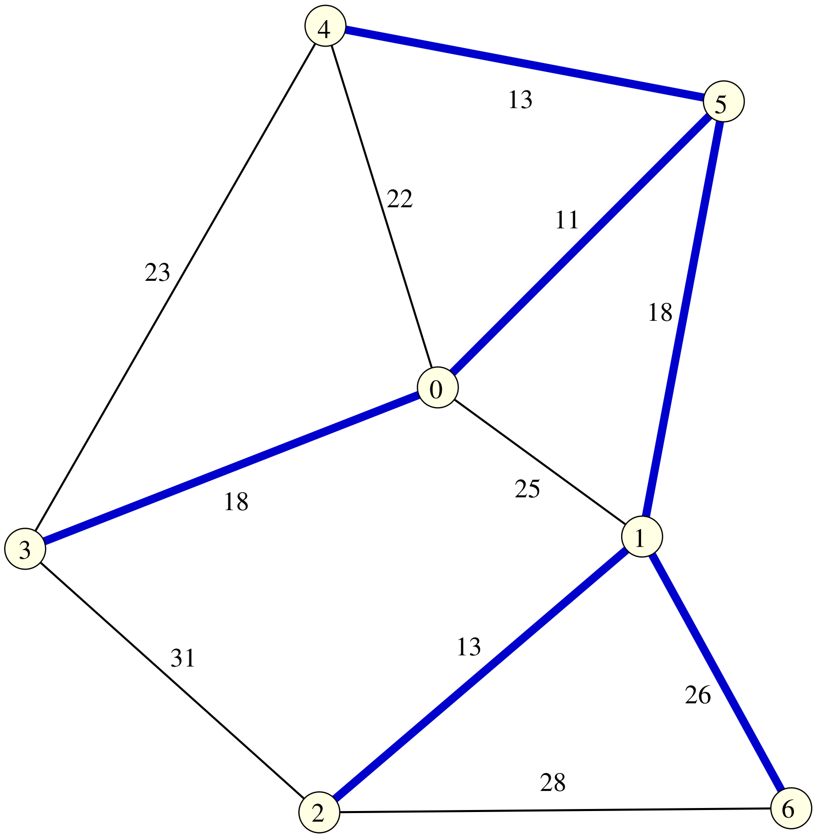

Figure 6.8 A minimum spanning forest in a graph G. The edges in the minimum are indicated in bold. The cost of each edge is indicated. This figure was generated with the spanning tree demo in xlman.

p. If $j = i$ we are done. So assume $j > i$ and consider $F' = F_0 \setminus e_j \cup e_i$. The cost of F' is at most the cost of F_0, F' is a spanning forest (since the removal of e_j splits one component of F_0 into two components each containing one of the endpoints of e_i and hence the addition of e_i glues them together again), and F' is lexicographically smaller than F_0, a contradiction. Thus $j = i$.

In an implementation the crucial question is how to check whether an edge e should be added to T. The data type *node_partition* is ideally suited for that purpose. We maintain the connected components of T as a node partition P, i.e., two nodes of G belong to the same block of P iff they are connected by a path of edges of T. Then an edge $e = \{u, v\}$ closes a cycle with respect to T iff u and v belong to the same block of P, i.e., if $P.same\_block(u, v)$. If e does not close a cycle we add e to T and update P by uniting the blocks containing u and v ($P.union\_blocks(u, v)$). We obtain the following algorithm:

⟨*Kruskal.c*⟩≡
```
#include <LEDA/graph.h>
#include <LEDA/node_partition.h>
list<edge> MIN_SPANNING_TREE(const graph& G,
                             int (*cmp)(const edge&, const edge&))
{
  list<edge> T;
  node_partition P(G);
  list<edge> L = G.all_edges();
  L.sort(cmp);
  edge e;
  forall(e,L)
  { node u = source(e);
    node v = target(e);
    if (! P.same_block(u,v))
    { T.append(e);
      P.union_blocks(u,v);
    }
  }
  return T;
}
```

The running time of Kruskal's algorithm is $O((n+m)\log(n+m))$, where m is the number of edges of G, since it takes time $O(m \log m)$ to sort the edges by cost and since the *forall_edges*-loop has cost $O((n+m)\alpha(n)) = O((n+m)\log(n+m))$. Kruskal's algorithm is efficient, but there are asymptotically more efficient algorithms known. In particular, there is a randomized algorithm with linear running time [KKT95].

The algorithm in LEDA combines Kruskal's algorithm with a heuristic and works in three phases. In the first phase it selects the $3n$ cheapest edges and runs Kruskal's algorithm on them. This yields a forest T. In the second phase it goes through the remaining edges and discards all edges that do not connect distinct components of T; this amounts to a *same_block* operation for each edge. In the third phase the still remaining edges are sorted by cost and are considered for inclusion in T in order of increasing cost. The hope underlying this heuristic is that the $3n$ edges selected in the first phase will already form a large part of the spanning tree and hence most remaining edges are discarded in the second phase. A saving results since the edges discarded in the second phase do not have to be sorted. In particular, if the third phase is empty the running time is $O((n+m)\alpha(n))$.

Table 6.2 shows some running times of the minimum spanning tree algorithm.

Exercises for 6.8

1 Experiment with the following modification of Kruskal's algorithm. First select the cn edges of smallest cost for some small constant c, say $c = 3$. Run Kruskal's algorithm on them. Then scan through the remaining edges and discard all edges that close a cycle. Sort the remaining edges in order of increasing cost and proceed with Kruskal's algorithm.

2 Implement Prim's minimum spanning tree algorithm. Let G be a connected graph and

n	m	Time
25000	250000	2.843
50000	500000	6.414
100000	1000000	13.83

Table 6.2 Running time of minimum spanning tree algorithm: For each n and m we generated 10 random graphs with n and m edges and random edge weights in [0 .. 100000] and ran MIN_SPANNING_TREE on them. You may perform your own experiments by running the minspantree_time demo.

let s be an arbitrary node of G. Prim's algorithm grows a minimum spanning tree from s. It maintains a subset S of the nodes of G and a set T of edges that comprise a minimum spanning tree of S. Initially, $S = \{s\}$ and $T = \emptyset$. For each node $v \notin S$ let $dist(v)$ be the smallest cost of an edge connecting v to a node in S. In each iteration Prim's algorithm selects the node $v \notin S$ with the smallest *dist*-value and adds it to S. What is an appropriate data structure for the *dist*-values and how can the *dist*-values be updated upon the addition of a node to S?

3 Implement *node_partitions*.

6.9 Graph Generators

Constructing graphs by a sequence of *new_node* and *new_edge* operations is a boring process, at least for humans. LEDA offers some *graph generators*.

```
complete_graph(graph& G, int n);
```

makes G the complete graph on n nodes. A graph G is *complete* if for every pair (v, w) of distinct nodes there is an edge e with $source(e) = v$ and $target(e) = w$. A complete graph on n nodes has $n(n-1)$ edges.

```
random_graph(graph& G, int n, int m, bool no_anti_parallel_edges,
             bool loopfree, bool no_parallel_edges);
```

makes G a random graph with n nodes and m edges in the so-called $G_{n,m}$-model of random graphs. A graph in this model consists of n nodes and m random edges. A random edge is generated by selecting a random element from a candidate set C defined as follows:

- C is initialized to the set of all n^2 pairs (v, w) of nodes, if *loopfree* is false, and to the set of all $n(n-1)$ pairs of distinct nodes, if *loopfree* is true.
- Upon selection of a pair (v, w) from C the pair is removed from C, when *no_parallel_edges* is true, and the reversed pair (w, v) is removed from C, when *no_anti_parallel_edges* is true.

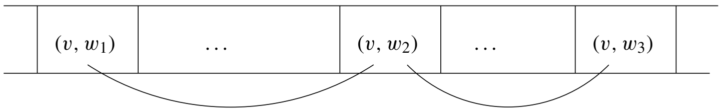

Figure 6.9 The storage layout of a graph generated by *random_graph_noncompact*. Memory is indicated as a horizontal band with low addresses at the left and high addresses at the right. Observe that the edges contained in any adjacency list spread over a large area of memory.

Several special cases of *random_graph* are available. The following pairs of calls are equivalent:

```
random_graph(G,n,m);
random_graph(G,n,m,false,false,false);

random_simple_graph(G,n,m);
random_graph(G,n,m,false,false,true);

random_simple_loopfree_graph(G,n,m);
random_graph(G,n,m,false,true,true);

random_simple_undirected_graph(G,n,m);
random_graph(G,n,m,true,true,true);
```

We give two implementations of *random_graph*. The first implementation works only for the case that all flags are set to false. The second implementation is to be preferred and we give the first implementation mainly for didactic reasons. The first implementation makes n calls of *new_node* and then m calls of *new_edge*(v, w) for random nodes v and w.

⟨*random_graph.c*⟩+≡

```
void random_graph_noncompact(graph& G, int n, int m)
{
  node*  V = new node[n];
  int i;
  G.clear();
  for(i=0; i<n; i++) V[i] = G.new_node();
  for(i = 0; i < m; i++)
    G.new_edge(V[rand_int(0,n-1)],V[rand_int(0,n-1)]);
  delete[] V;
}
```

Figure 6.9 indicates the storage layout generated by *random_graph_noncompact*. The edges are stored in the order in which they are generated. This implies that the edges belonging to any particular adjacency list are spread over a large area of memory and hence makes the layout not well suited for the most frequent iteration statement in graph algorithms: the iteration over the edges out of a node. A compact layout, which stores for each node all edges out of the node consecutively, is much better. A quantitative comparison will be given later in the section.

We turn to the function *random_graph_compact* that generates a representation where all edges out of any node are stored consecutively. It also supports the flags *no_anti_parallel_edges*,

...	(v, w_1)	(v, w_2)	(v, w_3)	...

Figure 6.10 The storage layout of a graph generated by *random_graph_compact*. Memory is indicated as a horizontal band with low addresses at the left and high addresses at the right. Observe that the edges contained in any adjacency list are stored next to each other.

loopfree, and *no_parallel_edges*. In the generation process we distinguish cases according to whether the candidate set C is modified during the generation process or not.

We first deal with the simple case that the candidate set C is not modified by the process. We choose the edges in two phases. In the first phase we choose the source node of each edge and hence determine the out-degree of each node. In the second phase we iterate over the nodes of the graph and generate for each node the required number of outgoing edges. In this way all edges out of a node are generated consecutively. The running time is $O(n+m)$.

⟨*random_graph.c*⟩+≡

```
void random_graph_compact(graph& G, int n, int m,
                          bool no_anti_parallel_edges,
                          bool loopfree, bool no_parallel_edges)
{ if ( n == 0 && m > 0 )
    error_handler(1,"random graph: m to big");
  if ( n == 1 && m > 0 && loopfree )
    error_handler(1,"random graph: m to big");
  node*  V = new node[n];
  int* deg = new int[n];
  int i;
  G.clear();
  for (i = 0; i < n; i++) { V[i] = G.new_node();
                            deg[i] = 0;
                          }
  if ( !no_anti_parallel_edges && !no_parallel_edges )
  {
    for (i = 0; i < m; i++) deg[rand_int(0,n-1)]++;
    for (i = 0; i < n; i++)
    { node v = V[i];
      int  d = deg[i];
      while ( d > 0 )
      { int j = rand_int(0,n-1);
        if ( loopfree && j == i ) continue;
        G.new_edge(v,V[j]);
        d--;
      }
    }
  }
  else { ⟨random graph: difficult case⟩ }
```

```
  delete[] V;
  delete[] deg;
}
```

We come to the case where the candidate set C is modified during the generation process. In this situation we have to work harder.

We first check whether m is too large. If only parallel edges are forbidden then m can be at most n^2, if parallel edges and self-loops are forbidden then m can be at most $n(n-1)$, if parallel and anti-parallel edges are forbidden then m can be at most $n + n(n-1)/2$, and if parallel edges and anti-parallel edges and self-loops are forbidden then m can be at most $n(n-1)/2$.

For the generation process we maintain a *node_map2<bool>* C with the following properties:

- If *loopfree* is false then $C(v, w) = true$ iff $(v, w) \in C$.
- If *loopfree* is true then for all v and w with $v \neq w$: $C(v, w) = true$ iff $(v, w) \in C$, i.e., the map C is equal to the set C except on the diagonal. This relaxed "equality" removes the obligation to set $C(v, v)$ to false for all v.

We build the graph as follows. We generate a random pair (v, w) of nodes. If it does not belong to the candidate set, we discard it, and if it belongs to the candidate set, we add it to the graph and update the candidate set accordingly. We build the graph temporarily as an array E of lists of nodes. Once we have constructed all edges of the graph in E we actually construct G.

⟨*random graph: difficult case*⟩≡

⟨*random graph: check whether m is too big*⟩

```
node_map2<bool> C(G,true);
array<list<node> > E(n);
int i = m;
while ( i > 0 )
{ int vi = rand_int(0,n-1);
  node v = V[vi];
  node w = V[rand_int(0,n-1)];
  if ( (v == w && loopfree) || !C(v,w) ) continue;
  E[vi].append(w);
  if ( no_parallel_edges ) C(v,w) = false;
  if ( no_anti_parallel_edges ) C(w,v) = false;
  i--;
}
for (i = 0; i < n; i++)
{ node v = V[i];
  node w;
  forall(w,E[i]) G.new_edge(v,w);
}
```

Random	Simple	Simple loopfree	Simple undirected
10.97	21.05	20.98	24.24

Table 6.3 Running time of random graph generation: We generated a random graph with $n = 10^5$ nodes and $m = 10^6$ edges. The first column shows the running time with all flags set to false, and the other columns show the time to generate a simple graph, a simple loopfree graph, and a simple undirected graph, respectively. You may perform you own experiments using the random graph demo.

What is the running time of the generation process? The less mathematically inclined reader may skip the remainder of this section. We do the analysis for the case that no parallel edges are allowed and leave the other cases to the reader. In this situation the maximal number of edges is $M = n^2$ and each edge generated decreases the number of candidate edges by one. Thus there are $M - j$ candidate edges when j edges have already been generated, and hence an expected number of $M/(M - j)$ iterations are needed to generate a candidate. We conclude that the expected total number of iterations required to generate m edges is

$$\sum_{0 \le j < m} M/(M - j).$$

If $m > M/2$ this sum is less than (we use the estimate $\sum_{1 \le j \le k} 1/j \approx \ln k$)

$$2m \sum_{M-m+1 \le j \le M} 1/j = O(m(\ln M - \ln(M - m))) = O(m \ln(M/(M - m)))$$

and if $m < M/2$ this sum is $O(m)$. In either case the running time is $O(m(1 + \ln(M/(M - m))))$.

We still need to implement the check of whether m is too big. This check is non-trivial to implement due to the danger of overflow. Note that n^2 may be a number which does not fit into an *int*. We therefore cannot simply compute the upper bound for the number of edges in a variable of type *int*. We use a variable of type *double* instead. This will work as long as $n \le 2^{26}$, which is safe for some time to come. We only show one case of the check.

⟨*random graph: check whether m is too big*⟩≡
```
double md = m; double nd = n;
if ( no_parallel_edges && !loopfree &&
      !no_anti_parallel_edges && md > nd*nd)
  error_handler(1,"random graph: m too big");
```
⟨*random graph: more checks whether m is too big*⟩

Table 6.3 shows the running time of our random graph generators.

The storage representation of a graph can have significant impact on the running time of graph algorithms. We give an example. We generate a random graph with either one of the

n	m	Compact	Non-compact
100000	1000000	0.34	0.85

Table 6.4 Influence of representation on running time: We generated a random graph with n nodes and m edges with our two random graph generators and then ran ⟨*determine number of edges*⟩ on both of them. Observe that the running time is more than double for the non-compact representation. You may perform your own experiments by running the compact_versus_noncompact_representation demo.

two generators above and then count the number of edges in the graph by iterating over all the edges out of all nodes.

⟨*determine the number of edges*⟩≡

```
count = 0;
forall_nodes(v,G)
  forall_adj_edges(e,v) count++;
```

Table 6.4 shows the running times for the compact and the non-compact representation. The difference is huge. The running time for the non-compact representation is more than double the running time for the compact representation. Similar but not as striking differences can be obtained for other graph algorithms. The effect is less pronounced for other graph algorithms because they usually do more than incrementing a counter in the *forall_adj_edges*-loop.

The difference in speed is due to the influence of cache memory. It makes access to consecutive locations faster than access to random locations. We discuss the influence of cache memory on running time in some detail in Section 3.2.2.

In earlier versions of LEDA we used *random_graph_noncompact* as our random graph generator. When we moved to *random_graph_compact* the running time of all our graph algorithms improved significantly.

```
random_graph(graph& G, int n, double p);
```

makes G a random graph with n nodes and an expected number of $p \cdot n \cdot (n-1)$ edges. The graph is generated by the following experiment. First n nodes are created and then for any pair (v, w) of distinct nodes the edge (v, w) is added to G with probability p. In the graph literature this model of random graphs is called the $G_{n,p}$-model. The running time is $O(n^2)$. Graphs generated according to the $G_{n,p}$-model behave similar to graphs generated according to the $G_{n,pn(n-1)}$-model.

```
random_bigraph(graph& G, int a, int b, int m,
               list<node>& A, list<node>& B);
```

makes G a random bipartite graph with a nodes on the one side, b nodes on the other side, and m edges directed from the A-side to the B-side. The nodes on the two sides are returned in A and B.

The generators for planar graphs are treated in the chapter on embedded graphs, see Section 8.9.

Exercises for 6.9

1 Compare the compact and the non-compact representation of graphs for other graph algorithms.

2 Let $o = (o_0, \ldots, o_{n-1})$ and $i = (i_0, \ldots, i_{n-1})$ be vectors of non-negative integers with $\sum_{0 \le j < n} o_j = \sum_{0 \le j < n} i_j$. Show that there is a graph with n nodes and o_j edges out of node j and i_j edges into node j for all j, $0 \le j \le n-1$. Generate a random graph of this kind. Hint: Use the class *dynamic_random_variate* of Section 3.5. Set up random variates S and T according to the weight vectors o and i, respectively. Use S to choose sources and T to choose targets. After every generation of an edge decrement the weight of its source and its target.

3 Use *random_graph*(G, n, m) to generate a random graph and test the graph for simplicity (using *Is_Simple*(G)). Try to find the value of m (in relation to n) where about 50% of the generated graphs are simple. If you want to understand the experiment, read up on the so-called Birthday paradox, see for example [Fel68] or [MR95].

4 Write a $O(n + m)$ generator for random graphs in the $G_{n,p}$-model. Hint: Reduce the problem to generating a graph in the $G_{n,m}$-model. Let p_m be the probability that a random graph in the $G_{n,p}$-model has m edges. Show that the probability is maximal for $m \approx pn(n-1)$ by considering the quotient p_m/p_{m+1}. Also show that the probability falls off quickly as one goes away from $m \approx pn(n-1)$. The idea is now to generate m according to the distribution given by the p_m's and to call *random_graph*(G, n, m) afterwards. The problem with this approach is that the p_m's are numbers with long representations. A possible way around this problem is to write each p_m as a sum $p_{m,1} + p_{m,2} + \ldots$ where for each m the $p_{m,i}$ decrease exponentially in i. Consider the collection $\left\{ p_{m,i} \; ; 0 \le m \le n(n-1),\ i \ge 0 \right\}$ and order it approximately by size. Generate m according to this distribution and then call *random_graph*(G, n, m). Provide your solution as an LEP.

6.10 **Input and Output**

We discuss how to write graphs to a file (or standard output) and how to read graphs from a file. We support two formats, the format shown in Figure 6.11 (henceforth called the standard representation) and the GML-format [Him97]. We will not formally define either format.

```
G.write();
```

writes the standard representation of G on standard output.

```
G.write(string s);
```

writes G onto the file with name s and

```
LEDA.GRAPH
void
void
4
|{}|
|{}|
|{}|
|{}|
5
1 2 0 |{}|
1 3 0 |{}|
2 3 0 |{}|
2 4 0 |{}|
3 4 0 |{}|
```

Figure 6.11 The standard representation of the graph of Figure 6.1. In the case of a parameterized graph the node and edge labels are enclosed in the angular brackets.

```
G.write_gml(string s,...);
```

writes G in gml-format. The additional arguments of *write_gml* can be used to fine-tune the way nodes and edges are output.

```
G.read(string s);
G.read_gml(string s, ...);
```

read a graph G from the file with name s. Either the standard representation or the GML-representation is expected.

The following piece of code is useful during the debugging phase of a graph algorithm.

```
while (true)
{ generate G;
  G.write("graph.gw");
  run graph algorithm on G;
  check result and abort if incorrect;
}
```

If the program aborts, a witness that falsifies the algorithm can be found in the file with name graph.gw.

There are several ways to inspect the witness graph:

- One can visually inspect the file to which the graph was written. This is tedious even for very small graphs.
- One can load the graph into a graph window. This is the most convenient method and we give more details below.
- One can send it through a graph drawing algorithm, see Section 8.1, and display the result.

We give more details on how to load a graph into a graph window, see Chapter **??** for more information about the graphwin type. The following piece of code assumes that the graph written has an integer node label and an integer edge label and that a parameterized graph was used. We define a graph *GRAPH*<*int*, *int*> *G* and read it from the file. We then define a *GraphWin gw* for *G*. We tell *gw* that we want the so-called data labels of the nodes and edges displayed, we open the display and put *gw* into edit mode[4]. When this program is executed, a window will pop up in which the graph *G* is displayed. The nodes of *G* will appear at random positions. The layout can be modified by dragging nodes around.

⟨*simple_visualization.c*⟩≡

```
#include <LEDA/graphwin.h>
main()
{
  GRAPH<int,int> G;
  G.read("graph.gw");
  GraphWin gw(G);
  node v; edge e;
  gw.set_node_label_type(data_label);
  gw.set_edge_label_type(data_label);
  gw.display();
  gw.edit();
}
```

Actually, there is no need even to write the program above. Call any of the programs starting with “gw” in xlman and use the file menu to load the graph.

6.11 Iteration Statements

Iterating over the nodes and edges of a graph or all the edges incident to a particular node is an essential component of any graph algorithm. Accordingly, we have seen iteration statements already many times in this chapter. In this section we treat them in detail. We first give a precise definition of the semantics, then discuss the possibility of hiding and unhiding edges and the possibilities of changing the order of iteration, and finally discuss which modifications of a graph are legal during iteration.

6.11.1 *Basics*

In order to understand the iteration statements we need to learn a bit about the representation of graphs in LEDA. A graph is a collection of nodes and edges which are arranged into several lists:

- The nodes are arranged into a list of nodes.

[4] If the statement *gw.edit*( ) is omitted, the program will briefly flash the graph and then terminate.

- The edges are arranged into a list of edges.
- In directed graphs two lists of edges are associated with every node v:

$$adj\_edges(v) = \{e \in E \; ; \; v = source(e)\} ,$$

i.e., the list of edges starting in v, and

$$in\_edges(v) = \{e \in E \; ; \; v = target(e)\} ,$$

i.e., the list of edges ending in v. The list $adj\_edges(v)$ is called the adjacency list of node v. 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 node v. Here it contains all edges incident to v, i.e.,

$$adj\_edges(v) = \{e \in E \; ; \; v \in \{source(e), target(e)\}\} .$$

An undirected graph must not contain self-loops, i.e., it must not contain an edge whose source is equal to its target.

The semantics of the iteration statements for graphs now reduces to the semantics of the iteration statements for lists.

```
forall_nodes(v,G)       { }
forall_rev_nodes(v,G)   { }
```

iterate over the list of nodes in either forward or backward direction,

```
forall_edges(e,G)       { }
forall_rev_edges(e,G)   { }
```

iterate over the list of edges in either forward or backward direction,

```
forall_adj_edges(e,v)   { }
forall_out_edges(e,v)   { }
forall_in_edges(e,v)    { }
forall_inout_edges(e,v) { }
```

iterate over the lists $adj\_edges(v)$, $out\_edges(v)$, $in\_edges(v)$, and $out\_edges(v)$ followed by $in\_edges(v)$, respectively, and

```
forall_adj_nodes(u,v)   { }
```

iterates over the other endpoint, i.e., $G.opposite(v, e)$, of all edges e in $adj\_edges(v)$.

6.11.2 *Modification during Iteration*

The rules are simple:

- It is unsafe to modify an object while iterating over it.
- However, the item under the iterator can be removed from the object.

In our experience the exception covers most of the situations where one wants to perform modifications during an iteration.

The following piece of code iterates over the edges of a graph and deletes all edges whose cost is negative.

```
forall_edges(e,G) if ( cost[e] < 0 ) G.del_edge(e);
```

The following piece of code is an infinite loop as new edges are appended to the list of edges during iteration.

```
forall_edges(e, G) G.new_edge(G.target(e), G.source(e));
```

A safe way to add the reversal of every edge to G is to write:

```
list<edge> L = G.all_edges();
forall(e, L) G.new_edge(G.target(e), G.source(e));
```

6.11.3 ***Hiding and Restoring Edges***

Sometimes it is convenient to remove edges only temporarily from a graph. For this purpose we have the concept of a hidden edge.

```
G.hide_edge(e);
```

removes e temporarily from G until restored by

```
G.restore_edge(e);
```

The implementation is simple. *Hide_edge*(e) deletes e from G and stores it in a list of hidden edges and *restore_edge*(e) removes e from the list of hidden edges and puts it back into the list of real edges. The list of all hidden edges is available as *G.hidden_edges*( ), one can ask whether an edge e is hidden (*G.is_hidden*(e)), ...

The following lines of code hides all edges with negative cost, then runs some graph algorithm on the resulting graph, and finally restores all edges.

```
forall_edges(e,G) if ( cost[e] < 0 ) G.hide_edge(e);
// some graph algorithm
G.restore_all_edges();
```

The operations *hide_edge* and *restore_edge* change the order of the adjacency lists and hence should be used with *extreme care on embedded graphs*.

6.11.4 ***Rearranging Nodes and Edges***

The lists of nodes and edges may be arranged by sorting. There are many different ways to sort. We go through the possibilities for nodes and remark that a similar set of sorting routines exists for edges.

```
G.sort_nodes(int (*cmp)(const node&, const node&));
```

sorts the nodes according to the compare function *cmp* and

```
G.sort_nodes(const node_array<NT>& A);
```

sorts the nodes according to the values in the node array A (the type *NT* must be a number type). The running time of both functions is $O(n \log n)$.

```
G.sort_nodes(const list<node>& vl);
```

assumes that *vl* is a permutation of the nodes of G. This permutation is taken as the new node ordering. The running time is linear.

```
G.bucket_sort_nodes(int (*ord)(const node&));
```

uses bucket sort to sort the nodes according to the values of the function $ord(v)$. The running time is $O(n + (b - a + 1))$ where a and b are the minimal and maximal values of *ord*, respectively.

```
void bucket_sort_nodes(const node_array<int>& A);
```

uses bucket sort with the ordering function $ord(v) = A[v]$.

Sorting the set of nodes rearranges the list of nodes. Subsequent *forall_nodes* loops iterate over the nodes in the modified order.

Sorting the set of edges rearranges the list of edges and the adjacency lists of all nodes. Subsequent *forall_edges*, *forall_adj_edges* and *forall_out_edges* loops iterate over the nodes in the modified order.

For example, if *cost* is an edge array that assigns an integer or double valued cost to every edge, then

```
G.sort_edges(cost);
```

rearranges the list of all edges and also the adjacency lists of all nodes in order of increasing cost.

6.12 Basic Graph Properties and their Algorithms

We define some basic graph properties and give the algorithms that decide them. For some of the algorithms we give the implementation. Many of the functions discussed in this section are illustrated by Figure 6.12 and by the submenu “test” of menu “graph” of any xlman-demo starting with the characters “gw”.

6.12.1 *Functionality*

The function

```
void CopyGraph(GRAPH<node,edge>& H, const graph& G);
```

constructs an isomorphic copy H of G. For each node v of H the corresponding node in G is stored in $H[v]$ and for each edge e of H the corresponding edge of G is stored in $H[e]$. The mapping $v \longrightarrow H[v]$ is a bijection from the nodes of H to the nodes of G and for each

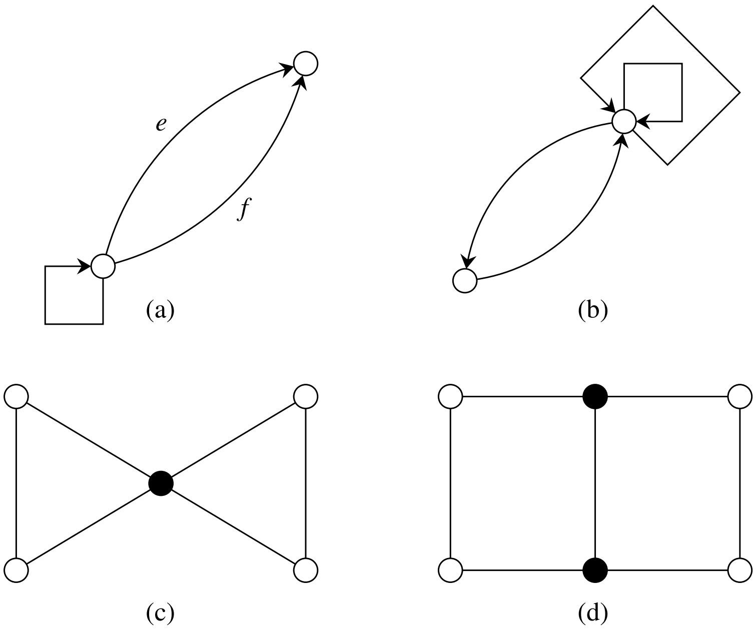

Figure 6.12 Illustration of basic graph properties: The graph (a) is not simple (the edges e and f are parallel) and has a self-loop h. The graph (b) is simple and bidirected. The graph (c) is connected but not biconnected (the full node is an articulation point). The graph (d) is biconnected but not triconnected (the full nodes form a split pair).

edge $e = (v, w)$ of H we have $source(H[e]) = H[v]$ and $target(H[e]) = H[w]$. We have already seen the implementation of *CopyGraph* in Section 6.1.

A graph is called *simple* iff is has no parallel edges, i.e., no two distinct edges e and f with the same source and sink, and a graph is called *loopfree* if it has no self-loop, i.e., no edge whose source is equal to its sink.

```
bool Is_Simple(const graph& G);
```

returns true if G is simple and returns false otherwise.

A directed graph $G = (V, E)$ is called *bidirected* if for every edge e the reversed edge $(target(e), source(e))$ also belongs to G, more precisely, if there is a bijection $rev : E \longrightarrow E$ such that:

- $source(e) = target(rev(e))$ and $target(e) = source(rev(e))$ for every $e \in E$ and
- $rev(e) \neq e$ for every $e \in E$.

The condition $rev(e) \neq e$ ensures that a self-loop cannot be its own reversal. A bidirected graph has an even number of edges. The main use of bidirected graphs is in the representation of embedded graphs, the topic of Chapter 8.

The calls

```
bool Is_Bidirected(const graph& G);
bool Is_Bidirected(const graph& G, edge_array<edge>& rev);
```

check whether G is bidirected. The second version also computes an appropriate bijection between the edges of G (if it exists).

```
void       Make_Bidirected(graph& G, list<edge>& R)
list<edge> Make_Bidirected(graph& G)
```

adds edges to G to make it bidirected. The added edges are returned in R or as the result of the function. An alternative to *Make_Bidirected* are the member functions *G.make_bidirected* and *G.make_map*( ) which are discussed in Section 8.2.

```
bool Is_Acyclic(const graph& G);
bool Is_Acyclic(const graph& G, list<edge>& L);
```

return true if the G is acyclic and return false otherwise. The second version also returns a list of edges whose removal makes G acyclic. We have already seen an implementation of the first version of *Is_Acyclic* in Section 6.2. The second version performs a depth-first search on G (see Section 7.3) and returns the list of back edges.

A *path in a directed graph* is a sequence

$$[v_0, e_1, v_1, e_2, v_2, \ldots, v_{k-1}, e_k, v_k]$$

of nodes and edges such that $source(e_i) = v_{i-1}$ and $target(e_i) = v_i$ for all i, $1 \le i \le k$. We call v_0 the source of the path and v_k the target of the path. The number of edges in the path is called the cardinality or length of the path. We will frequently abuse notation and write

$$[e_1, e_2, \ldots, e_k]$$

or

$$[v_0, v_1, v_2, \ldots, v_{k-1}, v_k]$$

instead of the more verbose notation above. A path is *simple* if all nodes (except maybe for the source and the target of the path) are pairwise distinct. A *cycle* is a path whose source is equal to its target.

A *path in an undirected graph* is a sequence

$$[v_0, e_1, v_1, e_2, v_2, \ldots, v_{k-1}, e_k, v_k]$$

of nodes and edges such that $\{source(e_i), target(e_i)\} = \{v_{i-1}, v_i\}$ for all i, $1 \le i \le k$ and $e_{i-1} \ne e_i$ for all i, $1 < i \le k$. We call v_0 and v_k the endpoints of the path. The number of edges in the path is called the cardinality or length of the path. We will frequently abuse notation and write

$$[v_0, v_1, v_2, \ldots, v_{k-1}, v_k]$$

instead of the more verbose notation above.

If G is a graph and e is an edge of G then $G \setminus e$ is the graph that results from removing e from G. If v is a node of G then $G \setminus v$ denotes the graph that results from removing v and all edges incident to v from G.

An undirected graph G is *connected* if for any two nodes v and w of G there is a path from v to w in G. An *articulation point* of an undirected graph G is any node of G such that $G \setminus v$ is not connected. An undirected graph is called *biconnected* if is has no articulation point. A *split pair* of an undirected graph is a pair $\{s_1, s_2\}$ of nodes such that $G \setminus \{s_1, s_2\}$ is not connected.

```
bool Is_Connected(const graph& G);
```

returns true if G (viewed as an undirected graph) is connected and returns false otherwise.

```
void       Make_Connected(graph& G,list<edge>& L);
list<edge> Make_Connected(graph& G);
```

make G connected by adding edges and return the list of inserted edges. The number of edges added is minimal.

```
void       Make_Biconnected(graph& G,list<edge>& L);
list<edge> Make_Biconnected(graph& G);
```

make G biconnected by adding edges and return the list of inserted edges.

```
bool Is_Biconnected(const graph& G);
bool Is_Biconnected(const graph& G, node& s);
```

test whether G is biconnected. The second version returns an articulation point in s if the graph is not biconnected.

A (directed or undirected) graph is *bipartite* if the nodes of the graph can be colored with two colors such that every edge of G connects nodes with different colors.

```
bool Is_Bipartite(const graph& G);
bool Is_Bipartite(const graph& G, list<node>& A, list<node>& B);
```

return true if G is bipartite and return false otherwise. The second version also returns a bipartition of the nodes of G in A and B (if the graph is bipartite).

A graph is *planar* if it can be drawn into the plane such that all nodes are placed at distinct points in the plane and such that no two edges cross.

```
bool Is_Planar(const graph& G);
```

returns true if G is planar and returns false otherwise. We will see a lot more of planar graphs in Chapter 8.

All functions above have linear running time $O(n + m)$.

```
bool Is_Triconnected(const graph& G);
bool Is_Triconnected(const graph& G, node& s1, node& s2);
```

returns true if G (viewed as an undirected graph) is triconnected and returns false otherwise. The second version returns a split pair in *s1* and *s2* if the graph is not triconnected. The running time is $O(n(n + m))$.

Table 6.5 reports some running times of the basic graph algorithms.

n	G	L	C	B	S	D	A	N	T
1000	0.07	0.01	0.01	0.03	0.04	0.1	0.01	0	17.9
10000	1.08	0.03	0.29	0.63	0.48	1.85	0.28	0.01	3342

Table 6.5 Speed of basic graph algorithms: We generated a random graph with n nodes and $m = 10n$ edges and then ran various graph algorithms on it:
G = generation of random graph,
L = time for removing self-loops,
C = time for testing connectedness,
B = time for testing biconnectedness,
S = time for testing simplicity,
D = time for testing bidirectedness,
A = time for testing acyclicity,
N = time for testing bipartiteness,
T = time for testing triconnectivity.
The time for testing bipartiteness is so small because a violation to bipartiteness is found very quickly in a random graph. For bipartite graphs the running time will be about the time to test connectedness. You may perform your own experiments by running the speed of basic graph algorithms demo.

6.12.2 *Implementations*

We give the implementation of the function *Is_Bidirected*.

We make two copies of the edges of G in lists *EST* and *ETS* and sort both lists.

In the sorted version of *EST* the edges are sorted by their source node, and edges with equal source node are sorted by their target node, i.e., all edges out of the first node come first, then all out of the second node, $\ldots$. Within each group of edges the ordering is by target node.

In the sorted version of *ETS* the edges are sorted by their target node, and edges with equal target node are sorted by their source node, i.e., all edges into the first node come first, then all into the second node, $\ldots$.

We use bucket sort for both sorts. This will play a role below.

Figure 6.13 shows an example. After having sorted the two lists the i-th edge of *EST* is the reversal of the i-th edge of *ETS* for all i (if G is bidirected).

Self-loops cause a small problem. As described so far, a self-loop can be matched with itself. There is a simple remedy. We use the fact that bucket sort is stable, i.e., the relative order of parallel edges is not changed.

Suppose now that we reverse *ETS* before the sorting step. Consider all self-loops incident to a particular node v, say $e_1, e_2, \ldots, e_k$. In *EST* they will appear exactly in the same order as in the original list of edges and in *ETS* they will appear in the reversed order. We match the i-th edge of one sequence with the i-th edge of the other sequence. When k is even we obtain a legal matching and when k is odd we will attempt to match one of the edges with itself. This leads to the following program.

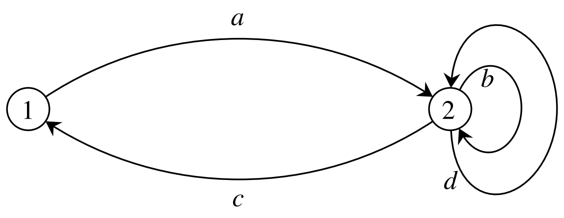

Figure 6.13 The lists *EST* and *ETS* in the implementation of Is_Bidirected: It is assumed that the original edge list of G is $E = (a, b, c, d)$. Observe that the edges b and d are parallel. In *EST* the edges are sorted by source, and edges with equal source are sorted by target. Parallel edges appear in the same order as in E. Thus $EST = (a, c, b, d)$. In *ETS* the edges are sorted by target, and edges with equal source are sorted by source. Parallel edges appear in the reverse order as in E. Thus $EST = (c, a, d, b)$.

The program uses the fact that the nodes of a graph are internally numbered and that *index*(v) returns the number of a node v.

```
static int edge_ord1(const edge& e) { return index(source(e)); }
static int edge_ord2(const edge& e) { return index(target(e)); }
bool Is_Bidirected(const graph& G, edge_array<edge>& reversal)
{
  int n = G.max_node_index();
  edge e,r;
  list<edge> EST = G.all_edges();
  EST.bucket_sort(0,n,&edge_ord2);
  EST.bucket_sort(0,n,&edge_ord1);

  list<edge> ETS = G.all_edges();
  ETS.reverse();                                //crucial
  ETS.bucket_sort(0,n,&edge_ord1);
  ETS.bucket_sort(0,n,&edge_ord2);
  // merge EST and ETS to find corresponding edges
  while (! EST.empty() && ! ETS.empty())
  { e = EST.pop();
    r = ETS.pop();
    if ( target(r) == source(e) && source(r) == target(e)
         && e != r )
       reversal[e] = r;
    else return false;
  }
  return true;
}
```

Exercises for 6.12

1 Give an implementation of the function *Is_Simple*. Use a *node_map2*.

2 Implement a function that tests whether a graph has a self-loop.
3 Implement the function *Make_Acyclic*. Read Section 7.3 first.
4 As above, but for function *Is_Connected*.
5 As above, but for function *Make_Connected*.
6 Provide a better implementation of the triconnectedness test. A linear time algorithm is described in [HT73]. Provide it as an LEP.

6.13 Parameterized Graphs

Parameterized graphs are another convenient way to associate information with the nodes and edges of a graph.

```
GRAPH<vtype,etype> G;
```

declares G as a parameterized graph and initializes G to the empty graph. With every node of G a variable of type *vtype* is associated and with every edge of G a variable of type *etype* is associated. The variables associated with nodes or edges can be accessed using array notation, i.e., $G[v]$ and $G[e]$ return the variables associated with node v and edge e, respectively. We have illustrated the use of parameterized types already in Section 6.1. We will see extensive use of parameterized graphs in the chapters on embedded graphs and on geometry. Here we want to discuss the relationship between parameterized graphs and graphs.

All operations defined on instances of the data type *graph* are also defined on instances of any parameterized graph type *GRAPH*<*vtype*, *etype*>, i.e., instances of a parameterized graph type can be used wherever an instance of the data type *graph* can be used, in particular, 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 (the adjacency lists) can be accessed. The node and edge entries are hidden.

The operations

```
node_array<vtype>& G.node_data()
edge_array<etype>& G.edge_data()
```

make the information associated with the nodes (edges) of G available as a node array (edge array) of type *node_array*<*vtype*> (*edge_array*<*etype*>). These operations are extremely useful when one wants to run a graph algorithm that requires a node or edge array as a parameter on a parameterized graph where one has stored the appropriate information in the nodes and edges, respectively. For example,

```
GRAPH<int,int> G;
node_array<edge> pred(G);
DIJKSTRA(G,G.first_node(), G.edge_data(), G.node_data(), pred);
```

runs Dijkstra's algorithm on G taking the edge data of G as the edge costs and storing the node distances in the nodes of G.

We have four different ways to associate information with the nodes, and similarly with the edges, of a graph in this section: node arrays, node data slots, node maps, and parameterized graphs. We use all four of them in our own work. We use parameterized graphs when the node information is an essential part of the graph. For example, we use the type *GRAPH<point, ...>* for graphs embedded into the plane; the position of any node v is given as $G[v]$. If the information is only temporarily associated with the node, as, for example, in a graph algorithm, we use node arrays and node maps. We use node maps for sparse arrays, where only a fraction of the nodes need an entry, and we use node arrays for dense arrays. We use node data slots, if speed is of utmost importance and node information is accessed many times and in random order, and we use standard node arrays otherwise. Standard node arrays are the most convenient and most widely used way to associate information with nodes.

6.14 Space and Time Complexity

Graphs are represented in their adjacency lists representation and hence the space requirement is $O(n + m)$, where n and m are the number of nodes and edges of the graph, respectively. Most operations on graphs take constant time except, of course, those which change or inspect the entire graph. The iterators take time proportional to the number of objects they iterate over, so *forall_edges*(e, G) takes time $O(m)$. We give some more information about the constant factors involved.

The space requirement of a *graph* or *GRAPH* with n nodes and m edges is $O(1) + 44m + 52n$ bytes, i.e., a graph with 10^4 nodes and 10^5 edges needs about 5 megabytes. For *GRAPH<T1, T2>* where an object of type *T1* or *T2* needs more than one word of storage one also has to account for the information associated with the nodes and edges. For example, a point requires 8 bytes and hence a *GRAPH<point, int>* requires an additional $8n$ bytes.

There is a trade-off between the space requirement of graphs and the functionality offered by them. We give some examples. Our graphs are fully dynamic, i.e., nodes and edges can be added and deleted at any time, and hence the adjacency information of every node is stored in a doubly linked list. For static graphs the adjacency information could be stored in an array. Our graphs support the dynamic addition of additional node and edge labels (in the form of node and edge arrays and maps) and hence every node or edge needs to have an integer index. This index could be saved if all node and edge labels have to be declared at the time of the construction of the graph.

We turn to running time. There is a large number of tables with running times of graph algorithms in this book. The tables prove that it is possible to solve problems on fairly large graphs using our algorithms. Moreover, the time bounds achieved by (most of) our algorithms are competitive with what other researchers report.

Exercises for 6.14

1 Implement a version of directed graphs where each node only knows about its outgoing edges but not about its incoming edges and where the adjacency lists are stored as singly linked lists and hence can only be traversed from front to rear. Make the graph class compatible with LEDA's graphs and provide it as an LEP.

2 Implement static directed graphs where all edges are stored in a single array, all edges in a adjacency list are stored consecutively, and each node has two pointers into the array, one to the first edge of its adjacency list, and one to the edge after the last edge of its adjacency list.

Bibliography

[Dij59] E.W. Dijkstra. A note on two problems in connection with graphs. *Num. Math.*, 1:269–271, 1959.

[Fel68] W. Feller. *An Introduction to Probability Theory and its Applications*. John Wiley & Sons, 1968.

[Him97] M. Himsolt. The graphlet system. *Lecture Notes in Computer Science*, 1190:233–??, 1997.

[HT73] J. E. Hopcroft and R. E. Tarjan. Dividing a graph into triconnected components. *SIAM Journal on Computing*, 2(3):135–158, ???? 1973.

[KKT95] David Karger, Philip N. Klein, and Robert E. Tarjan. A randomized linear-time algorithm for finding minimum spanning trees. *J. Assoc. Comput. Mach.*, 42:321–329, 1995.

[Kru56] J.B. Kruskal. On the shortest spanning subtree of a graph and the travelling salesman problem. In *Proceedings of the American Mathematical Society*, pages 48–50, 1956.

[MR95] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

Index