Graphs

11

Summary: *Graphs and their associated algorithms are widely used for the processing of problems in information science. A common problem for which graphs are suited is finding the shortest path between two given points. Another problem is calculating a minimal path that passes a number of points. This is an interesting problem for a carrier who has to deliver goods to a series of customers in different towns. Another typical application is the maximization of message or material throughput in a network. The components of the STL allow the construction of graphs for a multitude of applications and of a library of suitable fast algorithms. This chapter deals with the structure of a graph class on the basis of STL components and a selection of algorithms (shortest paths, topological sorting).*

A graph consists of a set of vertices and edges that connect two vertices. If an edge is assigned a direction, the graph is called *directed*, otherwise it is undirected. Figure 11.1 shows one directed and one undirected graph with five vertices and five edges each.

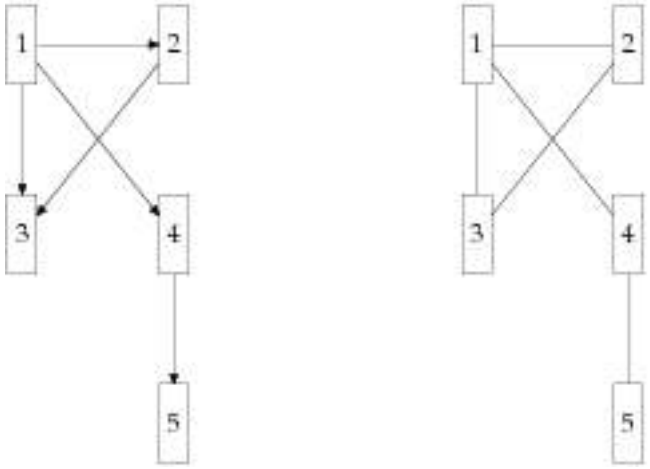


Figure 11.1: Directed and undirected graphs.

If an edge leads from a vertex A to a vertex B, A is called the *predecessor* of B and B is called the *successor* of A. A sequence of vertices e_1, e_2, ..., e_k is called a *path* if each vertex e_j with $j = 2$, ..., k is successor of vertex e_{j-1}.

There are different ways to represent a graph. The most frequently used representations are *adjacency matrices* (Latin *adiacere* = lie next to, border) and *adjacency lists*. In the adjacency matrix, a '1' at position (i, j) means that there is an edge from vertex i to vertex j. Each edge can be equipped with numbers which represent costs or distances. In this case, instead of the '1,' the corresponding numbers are entered. Furthermore, there must be one special number (normally 0) which means that no connection exists between the two vertices.

The adjacency matrix of an undirected graph is symmetric with regard to the main diagonal. Table 11.1 shows the adjacency matrices corresponding to Figure 11.1.

Vertex	1 2 3 4 5
1	0 1 1 1 0
2	0 0 1 0 0
3	0 0 0 0 0
4	0 0 0 0 1
5	0 0 0 0 0

Vertex	1 2 3 4 5
1	0 1 1 1 0
2	1 0 1 0 0
3	1 1 0 0 0
4	1 0 0 0 1
5	0 0 0 1 0

Table 11.1: Adjacency matrices for directed and undirected graphs.

The second common representation using adjacency lists consists of a vector or a list of all vertices, where for each vertex a sublist with all successive vertices exists (Figure 11.2).

This kind of representation has the advantage that only memory which is actually required is used and that, notwithstanding, the successors of each vertex can be found very quickly. For this reason, we will use the list representation, albeit in a slightly modified form.

Instead of taking lists for the references, as shown in Figure 11.2, the information about successors and edge values is stored in a map `map`. The key to an edge value is the number of a successive vertex. The advantage this gives over a list is that during the construction or reading of the graph, we can be certain that no multiple edges will exist. Thus, a vector element consists of a pair: the vertex and the set of its successors.

There is an alternative to this construction: imagine a graph as a map in which the set of successors and the edge values are accessed via a vertex, in analogy to the simple model of the sparse matrix on page 211. If the vertices are of type `string` and the edge values of type `double`, a graph type could be defined as follows:

```
typedef map<string, double> Successor;
typedef map<string, Successor> GraphType;
```

Now, the definition of vertices and edge values is very simple:

```
string vertex1("firstVertex");
string vertex2("secondVertex");
```

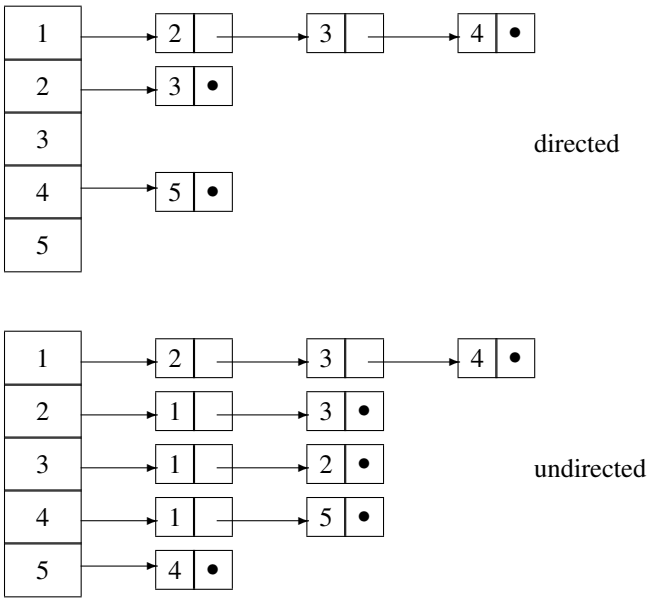


Figure 11.2: Adjacency lists.

```
GraphType theGraph;
theGraph[vertex1][vertex2] = 4.568;
```

This solution is not favored for various reasons:

- In many instances, the information about whether a graph is directed plays a role and should therefore be included.
- Inadvertent access to an undefined vertex with the `[]` operator leads to a new vertex being created without an error message (see page 79).
- At each access to a vertex, a search process is carried out. The process is fast ($O(\log N)$), but a direct access is even faster.
- Sometimes an order is needed, for example, the order of vertices in a shortest path. A vector of vertex numbers is a suitable and, above all, very simple tool for this purpose. Solutions based on the `GraphType` of the above listing are more expensive from a programming point of view.

The complexity of programs involving graphs is generally expressed in relation to the number of vertices and edges.

Edges without weighting?

Edges can be equipped with parameters to express distances or costs. In case *no* parameters are needed, an empty class with a minimum set of operations is defined as a placeholder:

```
struct Empty {
  public:
    Empty(int=0) {}
    bool operator<(const Empty&) const { return true;}
};

inline std::ostream& operator<<(std::ostream& os,
                                const Empty&) {
    return os;
}

inline std::istream& operator>>(std::istream& is, Empty& ) {
    return is;
}
```

With this class, it is possible to formulate a uniform class for graphs, together with auxiliary routines for input and output, valid for graphs with and without weighted edges.

11.1 Class Graph

According to Figure 11.2, the class `Graph` consists of a vector `V` of all vertices. As the advanced private part shows, the additional information about whether the graph is directed or not is present. An undirected graph is represented by the fact that for each edge, a second edge exists in the opposite direction. This takes up memory for what, in the final analysis, is redundant information, but has the advantage that each successor of an arbitrary vertex can be reached quickly.

The class is equipped with various checking methods whose diagnostic messages are output on the channel pointed to by the ostream pointer `pOut`.

```
template<class VertexType, class EdgeType>
class Graph {
  public:
    // public type interface
    typedef std::map<int, EdgeType> Successor;
    typedef std::pair<VertexType, Successor> vertex;
    typedef checkedVector<vertex> GraphType;
    typedef typename GraphType::iterator iterator;
    typedef typename GraphType::const_iterator const_iterator;

  private:
    bool directed;
```

```
  GraphType C;           // container
  std::ostream* pOut;

public:
  /*The following constructor initializes the output channel with cerr. A parameter
    must be specified as to whether the graph is directed or undirected, because this is
    an essential property of a graph.
  */
  Graph(bool g, std::ostream& os = cerr)
  : directed(g), pOut(&os) {
  }

  bool isDirected() const { return directed;}

  /*A graph is a special kind of container to which something can be added and whose
    elements can be accessed. Therefore, in the following typical container methods,
    their extents are limited to those needed for the examples. Thus, there is no method
    for explicit removal of a vertex or an edge from the graph.
  */

  size_t size() const  { return C.size();}
  iterator begin()     { return C.begin();}
  iterator end()       { return C.end();}

  // access to vertex i
  vertex& operator[](int i) {
     // the access is safe, because C is a checkedVector
     return C[i];
  }

  // addition of a vertex
  int insert(const VertexType& e);

  // addition of an edge between e1 and e2
  void insert(const VertexType& e1, const VertexType& e2,
              const EdgeType& Value);

  // addition of an edge between vertices no. i and j
  void connectVertices(int i,int j,const EdgeType& Value);

  /*The following methods are useful tools for displaying information on a graph and
    checking its structure. These methods are described in detail in the next sections.
  */
  // checking of a read data model
  // output on the channel passed to check()
  void check(std::ostream& = std::cout);

  // determine the number of edges
  size_t CountEdges();
```

```
    // determine whether the graph contains cycles
    // and in which way it is connected
    void CyclesAndConnectivity(std::ostream& = std::cout);
};      // Graph
```

The last method combines two tasks, because they can be carried out in a single run. The terms are explained in the description of the methods.

11.1.1 Insertion of vertices and edges

To avoid ambiguities, a vertex is entered only if it did not previously exist. The sequential search is not particularly fast; however, this process is needed only once during the construction of the graph.

```
template<class VertexType, class EdgeType>
int Graph<VertexType,EdgeType>::insert(
                            const VertexType& e) {
    for(int i = 0; i < size(); ++i)
       if(e == C[i].first)
          return i;

    // if not found, insert:
    C.push_back(vertex(e, Successor()));
    return size()-1;
}
```

An edge is inserted by first inserting the vertices, if they are needed, and by determining their positions. The edge construction itself is carried out by the function `connectVertices()`. It is passed the vertex numbers and, because there is no search procedure, it is very fast.

```
template<class VertexType, class EdgeType>
void Graph<VertexType,EdgeType>::insert(
                                  const VertexType& e1,
                                  const VertexType& e2,
                                  const EdgeType& Value) {
    int pos1 = insert(e1);
    int pos2 = insert(e2);
    connectVertices(pos1, pos2, Value);
}

template<class VertexType, class EdgeType>
void Graph<VertexType,EdgeType>::connectVertices(
             int pos1, int pos2, const EdgeType& Value) {
    (C[pos1].second)[pos2] = Value;

    if(!directed)   // automatically insert opposite direction too
      (C[pos2].second)[pos1] = Value;
}
```

11.1.2 Analysis of a graph

The method `check()` sets the output channel and calls all other checking methods.

```
template<class VertexType, class EdgeType>
void Graph<VertexType,EdgeType>::check(std::ostream& os) {
    os << "The graph is ";
    if(!isDirected())
        os << "un";

    os << "directed and has "
       << size() << " vertices and "
       << CountEdges()
       << " edges\n";
    CyclesAndConnectivity(os);
}
```

Determining the number of edges

Determining the number of edges of a given graph is simple: all that is required is to add the lengths of all adjacency lists. Undirected graphs are represented by two opposed edges for each connected pair of vertices; thus, in this case, the sum is halved.

```
template<class VertexType, class EdgeType>
size_t Graph<VertexType,EdgeType>::CountEdges() {
    size_t edges = 0;
    iterator temp = begin();

    while(temp != end())
         edges += (*temp++).second.size();

    if(!directed)
        edges /= 2;
    return edges;
}
```

Cycles, connection, and number of components

A graph has a *cycle* if there is a path with at least one edge whose first node is identical with the last node.

An undirected graph is *connected* if each vertex can be reached from each of the other vertices. For directed graphs, a distinction is made between a strong and a weak connection. A directed graph has a strong connection if a path exists from each vertex to each of the other vertices, that is, all vertices are mutually reachable. The connection is weak if an arbitrary vertex A is reachable from a vertex B, but not vice versa.

A graph is not connected if it is composed of two or more non-connected components. Figure 11.3 shows some examples.

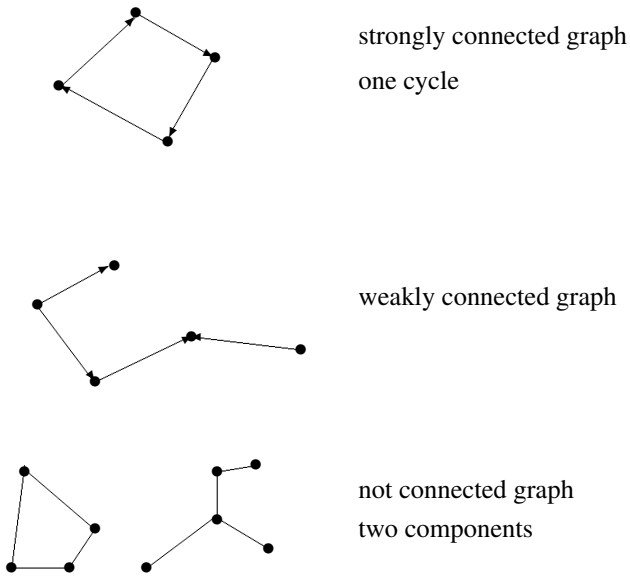


Figure 11.3: Different types of graph.

`CyclesAndConnectivity()` works with a *depth-first search*. Starting with an initial vertex, a successor is visited, then the successor of this successor, and so on, until no further successor is found. Then the next successor of the initial vertex is visited, applying the same process again. A breadth-first search, in contrast, processes all successors of the initial vertex in the first step, without considering their successors. Only then is the second level of successors tackled.

Unlike as suggested in Cormen *et al.* (1994), no recursion is employed because, depending on the system, even smaller graphs can cause a stack overflow. For undirected graphs, the stack depth corresponds to the number of edges + 1. In a user-defined stack, only the necessary information is stored, not all sorts of data used for function call management, such as local variables, return addresses, and so on.

```
template<class VertexType, class EdgeType>
void Graph<VertexType, EdgeType>::CyclesAndConnectivity(
                                        std::ostream& os) {
    int Cycles = 0;
    int ComponentNumber = 0;
    std::stack<int, std::vector<int> >
                    verticesStack; // vertices to be visited

    /*To prevent multiple visits to vertices in possible cycles, which entails the risk of
      infinite loops, the vertices are earmarked as having been visited or as finished
      being processed. This is executed by the vector VertexState.
    */
```

```
// assign all vertices the state 'not visited'
enum VertStatus {notVisited, visited, processed};
std::vector<VertStatus> VertexState(size(), notVisited);

/*If, starting from one vertex, an attempt is made to reach all other vertices, success
  is not guaranteed in weakly or non-connected graphs. Therefore, each vertex is
  visited. If it is found that a vertex has already been visited, it does not need to be
  processed any further.
*/

// visit all vertices
for(size_t i = 0; i < size(); ++i) {
  if(VertexState[i] == notVisited) {
     ComponentNumber++;
     // store on stack for further processing
     verticesStack.push(i);

     // process stack
     while(!verticesStack.empty()) {
        int theVertex = verticesStack.top();
        verticesStack.pop();
        if(VertexState[theVertex] == visited)
           VertexState[theVertex] = processed;
        else

           if(VertexState[theVertex] == notVisited) {
              VertexState[theVertex] = visited;
              // new vertex, earmark for processed mark
              verticesStack.push(theVertex);

              /*If one of the successors of a newly found vertex bears the
                visited mark, the algorithm has already passed this point
                once, and there is a cycle.
              */

              // earmark successors:
              typename Successor::const_iterator start =
                   operator[](theVertex).second.begin();
              typename Successor::const_iterator end   =
                   operator[](theVertex).second.end();

              while(start != end) {
                int Succ = (*start).first;
                if(VertexState[Succ] == visited) {
                     // someone's been here already!
                     ++Cycles;
                     (*pOut) << "at least vertex "
                             << operator[](Succ).first
                             << " lies in a cycle\n";
                }
```

```
                        /*Otherwise, the vertex has already been processed and
                          therefore should not be considered again, or it has not yet
                          been visited and is earmarked on the stack.
                        */

                        if(VertexState[Succ] == notVisited)
                           verticesStack.push(Succ);
                        ++start;
                     }
                  }
               }  // stack empty?
            }     // if(VertexState...
         }        // for() ...

         /*Now we only need the output. In case of directed, weakly connected graphs, the
           algorithm counts several components. To make the output conform to the above
           definitions, although with a lesser information content, a distinction is made as to
           whether the graph is directed or not.
         */

         if(directed) {
            if(ComponentNumber == 1)
               os << "The graph is strongly connected.\n";
            else
               os << "The graph is not or weakly connected.\n";
         }

         else
            os << "The graph has "
               << ComponentNumber
               << " component(s)." << std::endl;

         os << "The graph has ";
         if(Cycles == 0)
           os << "no ";
         os << "cycles." << std::endl;
      }
```

Display of vertices and edges

The output operator is used to display the vertices and edges of a graph. The output format corresponds to the format assumed by the routines of the next section.

```
template<class VertexType, class EdgeType>
std::ostream& operator<<(std::ostream& os, Graph<VertexType,
                         EdgeType>& G) {
   // display of vertices with successors
   for(size_t i = 0; i < G.size(); ++i) {
      os << G[i].first << " <";
```

```
        typename Graph<VertexType,EdgeType>::Successor::const_iterator
                  startN = G[i].second.begin(),
                  endN   = G[i].second.end();

        while(startN != endN) {
             os << G[(*startN).first].first  << ' ' // vertex
                << (*startN).second << ' ';         // edge value
             ++startN;
        }
        os << ">\n";
    }
    return os;
}
```

11.1.3 Input and output tools

This section presents some tools that facilitate experimenting with algorithms involving graphs. All auxiliary programs and sample data files are also available via the Internet.

Reading data

Besides information on the connection of vertices, many graphs need only the labelling of vertices and in some cases the length of edges. A simple way of representing this information in a file is the following format:

vertex < successor1 cost1 successor2 cost2 ...>

If they are not needed, the costs can be omitted. A # character at the beginning of a line starts a comment. Figure 11.4 corresponds to the simple file *gra1.dat*.

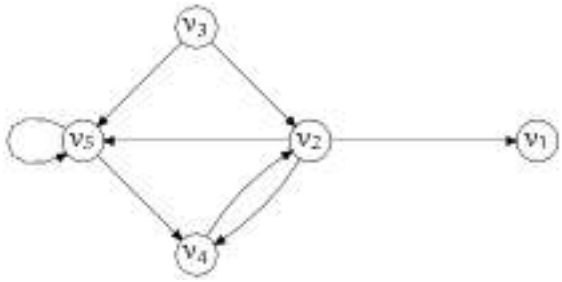


Figure 11.4: Directed graph. (Example taken from Maas (1994), by kind permission of Wißner publishers.)

```
# gra1.dat
v1
v2 <v1 v4 v5 >
```

```
v3 <v2 v5 >
v4 <v2 >
# cycle, loop to itself:
v5 <v5 v4 >
```

For the vertices, the graph needs only an identifier of type `string`. The edge parameters can be of a numeric type or, as in this example, of type `Empty` (see page 236). A program for reading and documenting a graph may then look as follows:

```
int main() {
    // no edge weighting, therefore type Empty:
    br_stl::Graph<std::string, br_stl::Empty> V(true);
                                  // true means directed

    br_stl::ReadGraph(V, "gra1.dat"); // file gra1.dat see above

    V.check();                    // display properties

    std::cout << V;               // display of vertices with successors
}
```

The result of method `check()` is:

The graph is directed and has 5 vertices and 8 edges.
The graph is not connected or is weakly connected.
The graph has cycles.

The display of the vertices with successors corresponds to the format of the input file. The function `ReadGraph()` is less interesting from an algorithmic point of view; it can therefore be found in Section A.1.2. A second example is an undirected graph with integer edge weights:

```
Graph<string,int> G(false);       // undirected
```

This graph is described by the following file, in which slight errors have been inserted for demonstration purposes:

```
# gra2.dat
v0 <v1 1 v5 3 >
#double edge v2
v1 <v2 5 v2 9 v4 3 v5 2 v0 1 >
v2 <v1 5 v5 2 v4 2 v3 1 >
v3 <v2 1 v4 3>
v4 <v5 1 v1 3 v2 2 v3 3 >
v5 <v1 2 v2 2 v4 1 >
```

The result of the above program, including output of the corrected graph with vertices and successors shown in Figure 11.5 is as follows:

The graph is undirected and has 6 vertices and 10 edges.
The graph has 1 component(s).
The graph has cycles.

v0 <v1 1 v5 3 >
v1 <v0 1 v2 5 v4 3 v5 2 >
v2 <v1 5 v3 1 v4 2 v5 2 >
v3 <v2 1 v4 3 >
v4 <v1 3 v2 2 v3 3 v5 1 >
v5 <v1 2 v2 2 v4 1 v0 3 >

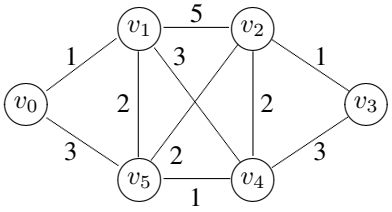


Figure 11.5: Undirected graph with weighted edges. (Example from Maas (1994).)

11.2 Dynamic priority queue

The STL provides the priority queue described in Section 4.3. For some purposes, the functions provided are not sufficient. For example, it is not possible specifically to change the priority of an element stored in a priority queue. Nor is removing and reinserting it with changed priority possible.

Exactly this property, however, is required in the algorithm for the topological sorting of a graph (see Section 11.3.2) and, furthermore, this property is advantageous in an algorithm for finding the shortest path from one node to another. This algorithm is described in Section 11.3.1. It could also be solved with a conventional STL priority queue, but only with a relatively higher number of push calls.

Since the required priority queue allows modification of stored elements without losing the priority queue property, this type will be called a 'dynamic priority queue.' It is intended as a special extension, so that it is not necessary to reproduce all the methods of an STL priority queue, but only those needed in this application.

At first sight, it seems a good idea to exploit the existing STL implementation. Two mechanisms are available:

- Inheritance
  The container used in the STL priority queue is `protected`, so that it can be accessed from within a derived class. However, the declaration would be very complex: the elements would be of type `pair<key, Index>`, with the priority being defined by the key `key` and the index representing a reference to the corresponding node of the graph. At the declaration, not only the underlying container, but also a comparison object `Greater` or something similar must be specified, since smaller keys are to signify a higher priority.

  Furthermore, the size of the additional code to be written is of the order of a whole new class, as experiments which are not documented here have shown.

- Delegation
  It is conceivable to invent a class that *uses* an STL priority queue by making it an attribute of this class and forwarding method calls to it. This possibility is ruled out because, owing to its `protected` property, the container cannot be accessed, but an access would be impossible to prevent.

Thus, it is more appropriate to write a special class. Even from the point of view of total cost of design, coding and testing, it is more advantageous than copying and complementing an existing implementation of a priority queue.

11.2.1 Data structure

The dynamic priority queue should allow an algorithm of the following kind:

1. Initialize the dynamic priority queue `DPQ` with a vector `V` which consists of the elements 0 ... $n-1$, and $n = V.size()$ holds.
2. As long as the `DPQ` is not empty:
   - determine from `DPQ` the element k of `V` that has the smallest value,
   - read the corresponding position `k` from `DPQ`,
   - modify one or more of the not yet read elements of `V`,
   - update `DPQ` accordingly.

Elements of `V` should be modified only via a dynamic priority queue method, because the information on the element to be modified must be kept. All this should also be *fast*, which excludes linear search processes or a reinitialization of the dynamic priority queue at each modification. To satisfy these requirements, the data structure shown in Figure 11.6 is chosen. In Figure 11.6, `c` is a vector of iterators to the elements of the external vector `V`. After initialization, `c` is converted into a heap with the property that `c[0]` now points to the smallest element inside `V`. After the heap conversion, the order of elements in `c` no longer corresponds to the order in `V`. To guarantee fast access in spite of this fact, an auxiliary array `Indices` is created with the necessary information, that is, element i of the array contains the address of array `c`, where the iterator to element i of vector `V` can be found (inverted addressing). This allows fast changes without search processes:

```
// modify element V[i] from within the dynamic priority queue:
*c[Indices[i]] = newValue;
```

At each modification of the heap `c`, the auxiliary array must be updated, which takes place in constant time.

11.2.2 Class dynamic_priority_queue

The heap inside the dynamic priority queue is indirect because it consists of iterators whose ordering obviously does not correspond to the iterators themselves, but to the

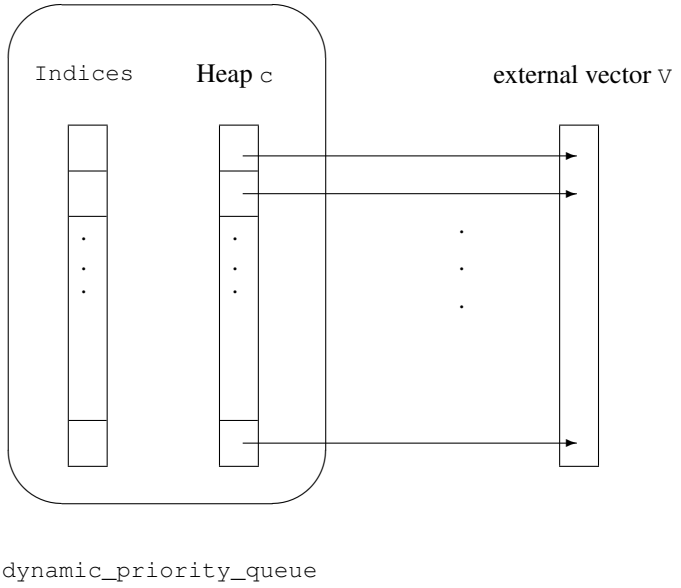


Figure 11.6: Internal data structure of the dynamic priority queue.

values pointed to by these iterators. The class `IterGreater` allows the creation of suitable function objects:

```
// compares the associated values of passed iterators
template<class T>
struct IterGreater {
    bool operator()( T x,  T y) const { return *y < *x;}
};
```

It should be noted that only the < operator is needed for the template data type `*T` and that the required relation is created by swapping the arguments. The class template `dynamic_priority_queue` needs only the type `key_type` of the elements of the external vector, which represent the priorities.

```
// include/dynpq.h
#ifndef DYNPQ_H
#define DYNPQ_H
#include<checkvec.h>
#include<algorithm>
#include<showseq.h>
namespace br_stl {

template <class key_type>
class  dynamic_priority_queue {
```

```
  public:
    // public type definitions
    typedef std::vector<key_type>::size_type size_type;
    typedef std::vector<key_type>::difference_type
                                                  index_type;

    // constructor
    dynamic_priority_queue(std::vector<key_type>& v);

    // change a value at position at
    void changeKeyAt(index_type at, key_type k);

    // index of the smallest element (= highest priority)
    index_type topIndex() const { return c.front()-first;}

    // value of the smallest element (= highest priority)
    const key_type& topKey() const { return *c.front(); }

    void pop();        // remove smallest element from the heap

    bool empty() const { return csize == 0;}
    size_type size() const { return csize;}

  private:
    checkedVector<index_type> Indices;      // auxiliary vector
    typedef typename std::vector<key_type>::iterator
                                       randomAccessIterator;
    checkedVector<randomAccessIterator> c;  // heap of iterators
    randomAccessIterator first;     // beginning of the external vector
    IterGreater<randomAccessIterator> comp; // comparison object
    index_type csize;                       // current heap size

    // heap update (see below)
    void goUp(index_type);
    void goDown(index_type);
};
}
```

The class definition is followed by the implementation together with explanations about the way of functioning. In the initialization list, the vectors `Indices` and `c` are created, among others. Subsequently, the addresses of all the elements of the external vector are entered and a heap is generated. An entry of the auxiliary array `Indices` is simply the difference between the address stored in `c` and the starting address of the vector `V`.

```
template <class key_type>
dynamic_priority_queue<key_type>::
                dynamic_priority_queue(vector<key_type>& v)
: Indices(v.size()), c(v.size()), first(v.begin()),
  csize(v.size()) {
```

```
    // store iterators and generate heap
    for(index_type i = 0; i< csize; ++i)
       c[i] = v.begin() + i;

    make_heap(c.begin(), c.end(), comp);          // STL

    // construct index array
    for(index_type i = 0; i< csize; ++i)
       Indices[c[i] - first] = i;
}
```

The method `changeKeyAt()` allows a value of the external vector at position `at` to be changed without violating the heap property. This process is of complexity $O(\log N)$ and therefore very fast. N is the number of elements still present in the heap. The main cost lies in the procedures for the reorganization of the heap which, however, never require more steps than $\log N$, the height of the heap.

The theory is that, if a modified element has become greater (= 'heavier'), then this element is allowed to sink down in the heap until it has reached its proper position. Vice versa, a 'lighter' element should rise by a corresponding amount towards the top.

```
template <class key_type>
void dynamic_priority_queue<key_type>::
                    changeKeyAt(index_type at, key_type k) {
   index_type idx = Indices[at];
   assert(idx < csize);    // value still present in the queue?

   if(*c[idx] != k) {      // in case of equality, do nothing
      if(k > *c[idx]) {
           *c[idx] = k;    // enter heavier value
           goDown(idx);    // reorganize heap
      }
      else {
           *c[idx] = k;    // enter lighter value
           goUp(idx);      // reorganize heap
      }
   }
}
```

The method call `goUp(idx)` causes an element to rise at position `idx`. Figure 11.7 shows, from top to bottom, the effect of `changeKeyAt()` and `goUp()`, starting with an arbitrary external vector whose ninth element is set to 0. The lighter element at `idx` rises through gradual sinking of the heavier predecessors and entry at the freed position.

```
template <class key_type>
void dynamic_priority_queue<key_type>::goUp(
                                              index_type idx) {
    index_type Predecessor = (idx-1)/2;
```

```
    randomAccessIterator temp = c[idx];

    /*In the figure, the process is exemplified by swapping the values of predecessor and
      successor. In the following program segment, however, in order to save unnecessary assignments, entry of the element temp (0 in the figure) is postponed until all the necessary exchange operations have been carried out.
    */
    while(Predecessor != idx
          && comp(c[Predecessor], temp)) {
       c[idx] = c[Predecessor];
       Indices[c[idx]-first] = idx;
       idx = Predecessor;
       Predecessor = (idx-1)/2;
    }

    c[idx] = temp;
    Indices[c[idx]-first] = idx;
}
```

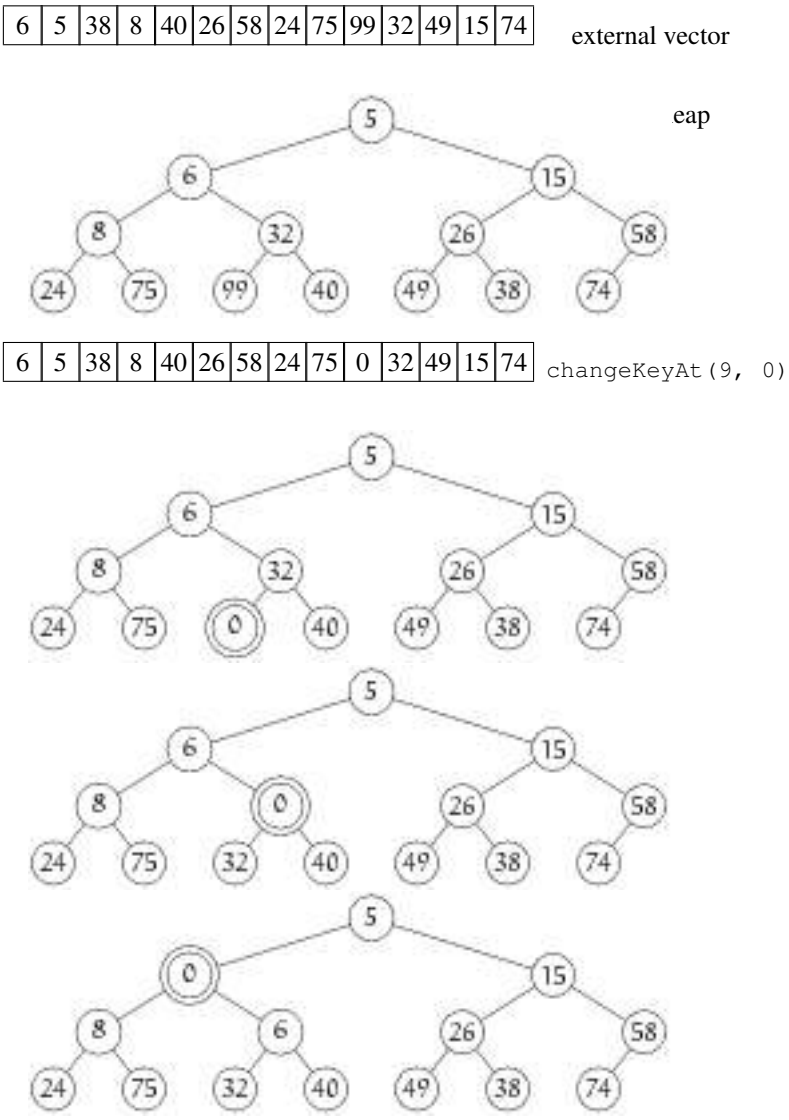
The method `goDown()` functions correspondingly. The heavy element at `idx` sinks down through gradual rising of the lighter successor and entry at the freed position.

```
template <class key_type>
void dynamic_priority_queue<key_type>::goDown(
                                              index_type idx) {
     index_type Successor = (idx+1)*2-1;
     if(Successor < csize-1
        && comp(c[Successor], c[Successor+1]))
           ++Successor;
     randomAccessIterator temp = c[idx];

     while(Successor < csize && comp(temp, c[Successor])) {
        c[idx] = c[Successor];
        Indices[c[idx]-first] = idx;
        idx = Successor;
        Successor = (idx+1)*2-1;

        if(Successor < csize-1
           && comp(c[Successor], c[Successor+1]))
              ++Successor;
     }
     c[idx] = temp;
     Indices[c[idx]-first] = idx;
}
```

The method `pop()` removes the topmost element from the heap. This is done by moving the last element to the top and blocking the freed position with `--csize`. Subsequently, the element sinks down to its proper position.




```
template <class key_type>
void dynamic_priority_queue<key_type>::pop() {
    // overwrite iterator at the top with the address of the last element
    c[0] = c[--csize];

    // enter the new address 0 at the position belonging
    // to this element in the auxiliary array
    Indices[c[0]-first] = 0;

    // let the element at the top sink to the correct position corresponding to its weight
    goDown(0);
```

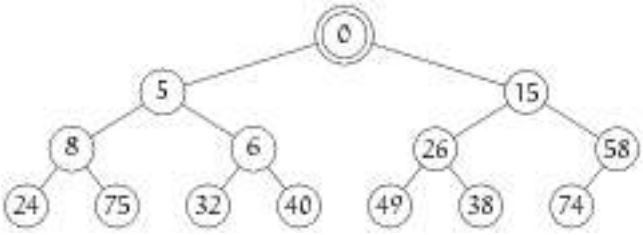


Figure 11.7: Effect of `changeKeyAt()` *and* `goUp()`.

```
}
#endif
```

Example

A program fragment shows the application:

```
// excerpt from k11/dynpq/maindpq.cpp
br_stl::checkedVector<double> V(8);

//... assign here values to the elements V[i]

br_stl::dynamic_priority_queue<double> DPQ(V);

// change value V[3]; correct insertion into DPQ is carried out automatically
DPQ.changeKeyAt(3, 1.162);

// outputting and emptying by order of priority
while(!DPQ.empty()) {
    std::cout << "index: " << DPQ.topIndex()
              << " value: " << DPQ.topKey() << std::endl;
    DPQ.pop();
}
```

11.3 Graph algorithms

There are vast numbers of algorithms for graphs. Here, only some of these are presented to show how such algorithms can be implemented using the STL and its techniques with the extensions of the previous sections.

Many problems involving graphs, such as finding the shortest path between two points or determining an optimal travelling route, involve specifying locations. For such problems, a vertex type suggests itself which contains the location's coordinates and a denomination. A suitable class for this is `Place`:

```
// include/place.h
#ifndef PLACE_H
#define PLACE_H
```

```
#include<cmath>
#include<string>

namespace br_stl {

class Place {
  public:
    Place() {};

    Place(long int ax, long int ay,
          std::string& N = std::string(""))
    : x(ax), y(ay), Name(N) {
    }

    const std::string& readName() const { return Name;}
    unsigned long int X()    const { return x;}
    unsigned long int Y()    const { return y;}

    bool operator==(const Place& P) const {
        return x == P.x && y == P.y;
    }

    // for alphabetical ordering
    bool operator<(const Place& P) const {
        return Name < P.Name;
    }

  private:
    long int x, y;                    // coordinates
    std::string Name;                 // identifier
};
```

Sometimes additional information, such as the number of inhabitants of a place, is required and can easily be added. The distance between two places can easily be calculated. The corresponding function `DistSquare()` is formulated as a separate function, because often only the result of a comparison of distances is of interest. To carry out the comparison, the squares of the distances are sufficient, and calculation of the square root `sqrt()` can be omitted.

```
inline unsigned long int DistSquare(const Place& p,
                                    const Place& q) {
   long int dx = p.X()-q.X();
   long int dy = p.Y()-q.Y();
   // (arithmetic overflow with large numbers is not checked)
   return dx*dx + dy*dy;
}

inline double Distance(const Place& p, const Place& q) {
    return std::sqrt(double(DistSquare(p,q)));
}
```

The output operator displays the name of the place and allows a simpler notation than the detour via `readName()`.

```
inline std::ostream& operator<<(std::ostream& os,
                                const Place& S) {
    return (os << S.readName());
}
} // namespace br_stl
#endif
```

11.3.1 Shortest paths

Here, the problem is to find the shortest path between two points of a graph. Probably the best known algorithm for this purpose is Dijkstra's algorithm. It uses the dynamic priority queue of Section 11.2. Figure 11.8 shows an undirected graph with 100 places, in which the shortest path between point 0 and point 63 is highlighted.

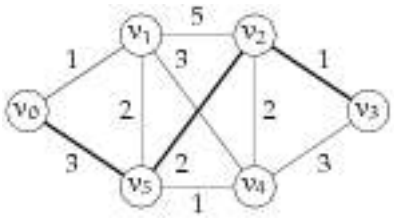


Figure 11.8: Graph with shortest path between two points.

The graph in Figure 11.8 was created with a number of small auxiliary programs. The function `create_vertex_set()` (Section A.1.3) can be used to generate a number of vertices with random coordinates within a given frame. The function `connectNeighbors()` (Section A.1.4) connects neighboring vertices of an undirected graph, and `createTeXfile()` (Section A.1.5) takes the graph and generates a file to be read into the LaTeX typesetting program used to typeset this book.

How should the `Dijkstra()` algorithm be used? In the following example, a graph `G` with random coordinates is constructed, but any other graph would do as well:

```
// excerpt from k11/dijkstra/mainplace.cpp
#include<gra_algo.h>        // contains Dijkstra()
#include<gra_util.h>        // auxiliary functions from Appendix A

using namespace std;

int main() {
    size_t Count = 100;
    br_stl::Graph<br_stl::Place,float> G(false); // undirected
    br_stl::create_vertex_set(G, Count, 12800, 9000); // range
```

```
    br_stl::connectNeighbors(G);

    /*The Dijkstra() function must be passed the graph, a vector of distances, and
      a vector of the predecessors, which are modified by the algorithm. The distance
      type must match the edge parameter type of the graph.
    */

    vector<float> Dist;
    vector<int> Pred;

    int start = 0;                  // starting point 0

    br_stl::Dijkstra(G, Dist, Pred, start);

    /*The last argument is the starting point which can be any vertex between no.
      0 and no. (G.size()-1). After the call, the distance vector contains the
      length of the shortest paths from each point to the starting point. Dist[k]
      is the length of the shortest possible path from vertex no. k to vertex no. 0.
      Dist[StartingPoint] is 0 by definition.
    */
    // output
    cout << "shortest path to "
         << G[start].first << ":\n";

    cout << "predecessor of:  is:  "
            "distance to [indices in ()]:\n";

    for(size_t i = 0; i < Pred.size(); ++i) {
       cout <<  G[i].first
             << '(' << i << ")    ";

        if(Pred[i] < 0)
           cout << "-";             // no predecessor of starting point
        else
          cout << G[Pred[i]].first;

        cout << '(' << Pred[i] << ')';
        cout.width(9);
        cout << Dist[i] << endl;
    }
}
```

The predecessor vector contains the indices of the predecessors on the path towards the starting point. `Pred[StartingPoint]` is undefined. If the starting vertex is 0, the predecessor and distance vectors of the graph in Figure 11.9 have the values shown in Table 11.2. It corresponds to the output of the above program, except that the table shows only the *vertex numbers* and not the *vertex names*.

The shortest path from vertex v_3 to vertex v_0 is 6 units long and leads through the vertices v_2 (=`Pred[3]`) and v_5 (=`Pred[2]`). The predecessor of 5 is 0. With this,

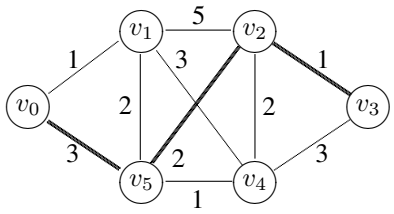


Figure 11.9: A shortest path.

i	Pred[i]	Dist[i]
0	undefined	0
1	0	1
2	0	3
3	2	5
4	1	4
5	3	6

Table 11.2: Example of predecessor and distance vectors.

the target is reached. There can be several equally short paths. The corresponding output of the program is:

shortest path to v0:

Predecessor of:	*is:*	*Distance to*	*[indices in ()]:*
v0(0)	*–(–1)*	*0*	
v1(1)	*v0(0)*	*1*	
v5(2)	*v0(0)*	*3*	
v2(3)	*v5(2)*	*5*	
v4(4)	*v1(1)*	*4*	
v3(5)	*v2(3)*	*6*	

How does the algorithm find the shortest path between two points? This algorithm is extensively described in several textbooks, e.g. Cormen *et al.* (1994). Therefore only a brief outline is given. A preliminary hint: below, the distance vector is to be initialized with the value ∞, which is 'approached' by the maximum value possible for the data type in question:

```
numeric_limits<aScalarType>::max() // maximum value
```

`aScalarType` is one of the basic data types `int`, `long`, `double`, and so on. The class `numeric_limits` is declared in the header `<limits>`.

The inclusion of the include files is followed by the definition of the `Dijkstra()` function which is passed the graph `Gr`, the two vectors of distances and predecessors, and the starting point of the search.

```
// include/gra_algo.h
#ifndef GRAPH_ALGORITHMS_H
```

```
#define GRAPH_ALGORITHMS_H
#include<dynpq.h>
#include<graph.h>
#include<limits>
#include<iostream>

namespace br_stl {

template<class GraphType, class EdgeType>
void Dijkstra(GraphType& Gr, std::vector<EdgeType>& Dist,
       std::vector<int>& Pred, int Start) {

    /*The algorithm proceeds in such a way that the distances are estimated and the
      estimates gradually improved. The distance to the starting point is known (0). For
      all other vertices, the worst possible estimate is entered.
    */

    Dist = std::vector<EdgeType>(Gr.size(),
           std::numeric_limits<EdgeType>::max());
                                          // as good as ∞
    Dist[Start] = (EdgeType)0;

    /*The predecessor vector too is initialized with 'impossible' values (–1). Subse-
      quently, a dynamic priority queue is defined and initialized with the distance vec-
      tor:
    */

    Pred = std::vector<int>(Gr.size(), -1);
    dynamic_priority_queue<EdgeType> Q(Dist);

    /*In the next step, all vertices are extracted one by one from the priority queue,
      and precisely in the order of the estimated distance towards the starting vertex.
      Obviously, the starting vertex is dealt with first. No vertex is looked at twice.
    */

    int u;                    // vertex with minimum
    while(!Q.empty()) {
       u = Q.topIndex();      // extract vertex with minimum
       Q.pop();

       /*Now, the distance estimates for all neighboring vertices of u are updated. If
         the previous estimate of the distance between the current neighbor of u and
         the starting vertex (Dist[Neighbor]) is worse than the distance between
         vertex u and the starting vertex (Dist[u]) plus the distance between u and
         the neighboring vertex (dist), the estimate is improved.

         This process is called relaxation. In this case, the path from the starting vertex
         to the neighbor cannot be longer than (Dist[u] + dist). In this case, u
         would have to be regarded as the predecessor of the neighbor.
       */
```

```
            // improve estimates for all neighbors of u
            typename GraphType::Successor::const_iterator
                        I = Gr[u].second.begin();

            while(I != Gr[u].second.end()) {
                int Neighbor = (*I).first;
                EdgeType dist = (*I).second;

                // relaxation
                if(Dist[Neighbor] > Dist[u] + dist) {
                    // improve estimate
                    Q.changeKeyAt(Neighbor, Dist[u] + dist);
                    // u is the predecessor of the neighbor
                    Pred[Neighbor] = u;
                }
                ++I;
            }
        }
    }
    // ... further graph algorithms (see later)
    } // namespace br_stl
    #endif
```

The loop cycles through all vertices. If the number of vertices is denoted by N_{V} and the number of edges by N_{E}, the complexity of the algorithm can be estimated as follows on the basis of the individual procedures:

1. N_{V} removals from the queue.
2. The removal (`pop()`) is of complexity $O(\log N_{\mathrm{V}})$.
3. Relaxation is carried out corresponding to the number of edges of a vertex. Since each vertex is looked at only once, its edges too are looked at only once. Therefore, a total of max. N_{E} edges are relaxed.
4. The relaxation of an edge is of complexity $O(\log N_{\mathrm{V}})$. The cost derives from reorganization of the heap in the method `changeKeyAt()`.

The removals 'cost' a total of $O(N_{\mathrm{V}} \log N_{\mathrm{V}})$, and the cost of all relaxations totals $O(N_{\mathrm{E}} \log N_{\mathrm{V}})$. Thus, the complexity of the whole algorithm is $O((N_{\mathrm{V}} + N_{\mathrm{E}}) \log N_{\mathrm{V}})$. In Cormen *et al.* (1994) it is demonstrated that the path found really is the shortest one. Obviously, there can be several equally short paths in a graph. Which of these is chosen depends on the arrangement of vertices and edges.

11.3.2 Topological sorting of a graph

Topological sorting is a linear ordering of all vertices of a graph in such a way that in the ordering each successive vertex appears *after* its predecessor.

One example is the references in an encyclopedia, in which a term is explained with the aid of other terms. A topological order of the terms would be an order in which references are made only to already defined terms.

A Gantt chart, in which it is determined which activity must be terminated before another one can be started, also contains topological sorting. Thus, when building a house, painters and decorators can start only when electricians and joiners have finished. These, in turn, can start only after the builders have erected the walls. A graph that describes such dependencies must not contain cycles. In other words, it cannot be that the builders can only erect the walls after painters and electricians have finished their work and that these, in turn, wait for the builders.

Some things can be done in arbitrary order, for example wall painting and installing the central heating boiler. Thus, different topological sortings of a graph are possible. A directed acyclic graph is often abbreviated to *dag* or *DAG*. Figure 11.10 shows a DAG which is not yet sorted topologically. The graph is defined by the following file with the structure known from page 243:

```
# topo.dat
a < l e f >
b < c e i >
c < f g j >
d < c g n >
e < h >
f < i m >
g < f >
h
i < h >
j < l k >
k < n >
l
m < j >
n
```

In the figure, activity a must be carried out before activity e. Activities h, l, and n are final activities; they have no successors and therefore stand at the end of the topological sorting shown in Figure 11.11. The dashed lines are redundant because the vertices can be reached via other edges. The essential difference is that all direction arrows point *to the right*. Vertices of the graph that have no topological precedence over each other are drawn more or less above each other. All vertices that have no predecessor are on the extreme left.

The algorithms in the quoted literature mostly proceed by way of a depth first search (Cormen *et al.* (1994)) or by complicated list structures which are constructed for the analysis (Wirth (1979)). In contrast to these, a method will be described which successively reads those vertices from a dynamic priority queue that have no predecessors, and then removes these vertices by updating the predecessor numbers of the other vertices. This method has the advantage of being very compact and very fast. Firstly, let us look at the program that calls the topological sorting:

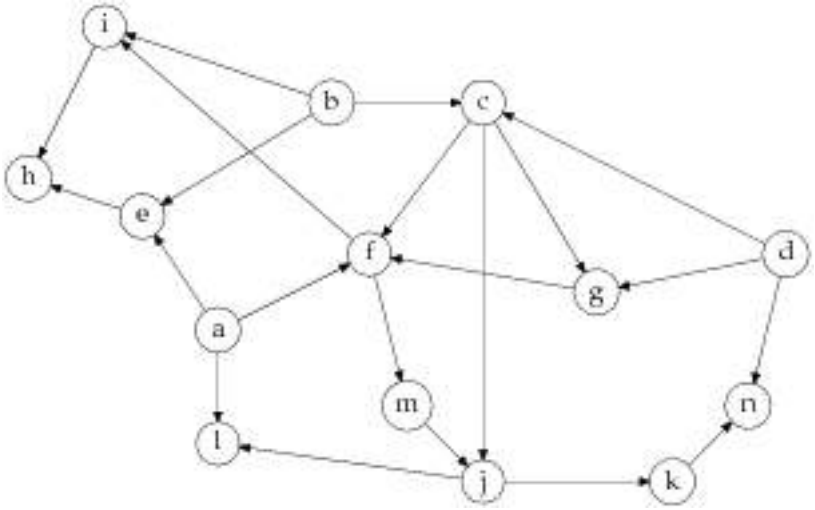


Figure 11.10: Directed acyclic graph.

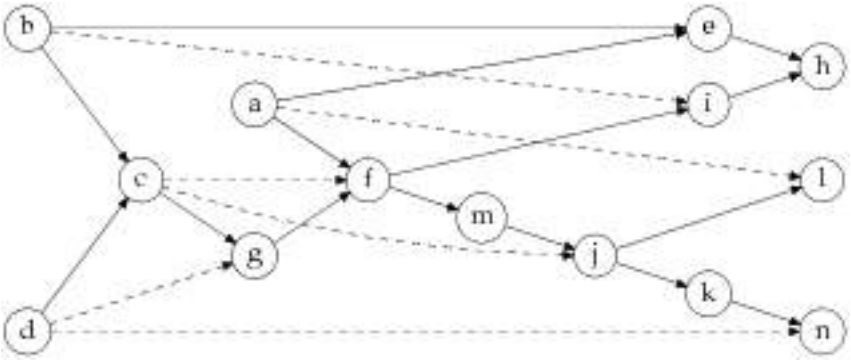


Figure 11.11: Topologically sorted DAG.

```
// k11/toposort/main.cpp : topological sorting
#include<gr_input.h>
#include<gra_algo.h> // contains topoSort(), see below
using namespace std;

int main() {
    br_stl::Graph<string, br_stl::Empty> G(true);   // directed
    br_stl::ReadGraph(G, "topo.dat");

    /*After sorting, the vector Ordering passed as argument contains the indices of
      the graph's vertices.
    */
```

```
    vector<int> Ordering;
    if(br_stl::topoSort(G, Ordering)) {          // sort
       for(size_t i = 0; i < G.size(); ++i)
          cout << G[Ordering[i]].first << ' ';
       cout << endl;
    }
    else cout << "Error in the graph!\n";
}
```

The output of the program corresponds to the representation in Figure 11.11:

d b a c e g f i m j k h n l

The algorithm proper follows. The function returns `false` if the graph contains one or more cycles. In such a case, the result is meaningless.

```
// File include/gra_algo.h (continued from page 258)
template<class GraphType>
bool topoSort( GraphType& G, std::vector<int>& Result) {
    assert(G.isDirected());            // let's play it safe!
    int ResCounter = 0;
    Result = std::vector<int>(G.size(), -1);

    /*The vector Result takes the indices of the correspondingly distributed vertices.
      The counter ResCounter is the position in Result where the next entry belongs.
    */

    checkedVector<int> PredecessorCount(G.size(), 0);
    int VerticesWithoutSuccessor = 0;

    /*For each vertex, the vector PredecessorCount counts how many predecessors it has. There are vertices without successors, whose number is kept in VerticesWithoutSuccessor. Furthermore, the algorithm remains stable if the precondition that a graph must not have cycles is violated. The variable VerticesWithoutSuccessor is used to recognize this situation (see below).
    */

    for(size_t iv = 0; iv < G.size(); ++iv) {

        if(G[iv].second.size() > 0) {   // is predecessor

           typename GraphType::Successor::const_iterator I =
                G[iv].second.begin();

           while(I != G[iv].second.end())
                // update number of predecessors
                ++PredecessorCount[(*I++).first];
        }
        else  { // Vertex is no predecessor, that is, without successor

              // an excessively high number of predecessors is used
              // for later recognition
```

```
            PredecessorCount[iv] =   G.size(); // too many!
            ++VerticesWithoutSuccessor;
        }
    }

    /*The dynamic priority queue is initialized with the vector of numbers of predecessors. At the beginning of the queue we find those vertices that have no predecessors and therefore are to be processed next. Only the vertices which are predecessors themselves, that is, that have successors, are processed. The subsequent loop is terminated when the queue contains only successor vertices which themselves are not predecessors. Their number of predecessors can never be 0 because earlier they were initialized with too high a value.
    */
    dynamic_priority_queue<int> Q(PredecessorCount);

    // process all predecessors
    while(Q.topKey() == 0) {
        // determine vertex with predecessor number 0
       int oV = Q.topIndex();
       Q.pop();

       Result[ResCounter++] = oV;

       /*To ensure that this vertex without predecessors oV is no longer considered in the next cycle, the number of predecessors of all its successors is decreased by 1.
       */
       typename GraphType::Successor::const_iterator
              I = G[oV].second.begin();
       while(I != G[oV].second.end()) {
          // decrease number of predecessors with
          // changeKeyAt(). Do not change directly!
          int V = (*I).first;
          Q.changeKeyAt(V, PredecessorCount[V] -1);
          ++I;
       }
    }

    /*Now, all vertices without successors are entered. As a countercheck, the variable VerticesWithoutSuccessor is decreased. If the queue contains too many vertices, an error message is displayed.
    */
    while(!Q.empty()) {
        Result[ResCounter++] = Q.topIndex();
        Q.pop();
        --VerticesWithoutSuccessor;
    }
```

```
    if(VerticesWithoutSuccessor < 0)
       std::cerr << "Error: graph contains a cycle!\n";
    return VerticesWithoutSuccessor == 0;
}
```

The error occurs when the graph contains at least one cycle, since in the cycle itself there can never be a vertex without a predecessor. In that case, more vertices are caught in the queue than there should be according to the number `VerticesWithoutSuccessor` counted at the beginning.

Complexity

For an estimate of the complexity, the following activities are relevant, where N_V is the number of vertices and N_E the number of edges. An auxiliary measure $n = N_E/N_\mathrm{V}$ is the average number of successors and predecessors per vertex:

1. Initialization of the vector with numbers of predecessors: $N_\mathrm{V} + N_\mathrm{E}$.
2. Initialization of the dynamic priority queue: N_V.
3. `while` loops: in all loops, each vertex is treated exactly once (N_V) and each edge (successor vertex) depending on the number of predecessors and successors (nN_E). Each 'treatment' means removal from the queue ($\log N_\mathrm{V}$) or modification of the queue with `changeKeyAt()` (again $\log N_\mathrm{V}$).

The dominating part is $N_\mathrm{V} \log N_\mathrm{V} + nN_\mathrm{E} \log N_\mathrm{V}$. If the number of vertices and edges is approximately the same, the cost to be expected is $O(N_\mathrm{V} \log N_\mathrm{V})$. The upper limit for the number of edges, however, is $N_\mathrm{V}(N_\mathrm{V} - 1)/2$ (every vertex is connected with every other vertex), so that the complexity is $O(N_\mathrm{V}^2 \log N_\mathrm{V})$.

Exercise

11.1 What happens if you run the program on page 260 with a file *topo.dat* in which the line `j < l k >` is substituted with `j < f l k >`?