Graph Interface Proxy Components Using CORBA

Shawn Bisgrove Jan Kratky Cory Plock

April 17, 1999

Contents

1 Introduction

This paper describes the implementation of a set of proxy components enabling the creation of a distributed graph system using CORBA. We do not offer here an introduction to CORBA; some understanding of CORBA concepts and implementation is assumed.

We describe and document only those classes not entirely generated by an IDL compiler. For an understanding of IDL-compiler-generated class files, see the Object Management Group's world wide web site [OMG], or the `MICO` literature at [MICO].

After a brief discussion of the concept of a "proxy component," we present the Interface Definition Language (IDL) code for our distributed components. This section will serve as a reference to implementors of clients that make use of these components.

Next, we document the implementation classes, after which we present example servers and several clients that interact with these servers. Our clients make a wide variety of calls via both the Static Invocation Interface (SII) and the Dynamic Invocation Interface (DII). A timing-test client provides a comparison of time taken for local calls with time taken for distributed calls to a faster machine. An interactive client demonstrates a node-mapping scheme that may be useful to future client implementors.

Finally, a makefile and startup script are provided.

For implementation documentation of the Graph package used in the below code, please see Appendix A. For documentation on graph-algorithm implementation, see Appendix B.

2 What is a Proxy Component?

We use the term `proxy component` to denote an object layer between a client and a server that attempts to reformat, parse, or convert communications.

The idea of an intermediary layer between an object and a complex subsystem is not new. This sort of scheme has been formalized as a design pattern, known as the `facade pattern`[Gamma95], in which a single contact point for a possibly complex system is developed, thereby simplifying client calls to that system. The decomposition of the problem in this fashion makes the system more modular and, hence, more manageable.

In a distributed setting, proxy components allow calls to be made to non-distributed code or *legacy code* without "cluttering" that code with the commands required for their network distribution. The resulting high modularity and loose coupling of system components permits easier coding and debugging—with the use of proxies, each component "stands on its own" to a greater degree.

It may be argued that adding proxy layers might slow system execution. However, without proxies, the data types manipulated within, and returned from, an object would be constrained to CORBA types. In C++, doing so limits the use of well-established specialized STL types and algorithms having documented stability and efficiency[Musser96]. Constituting an invitation to sloppy programming, such a CORBA-only approach quite easily could lead to programmer-introduced errors and inefficiencies.

The system's main IDL interfaces are `proxy_interface` and `proxy_factory`. The latter is responsible for creating instances of the former. This allows a single CORBA server to generate multiple instances of a `proxy_interface` for each of the clients. Without a construct like `proxy_factory`, all connecting clients would share the same `proxy_interface` instance (not always a desirable situation).

3 IDL Definitions/User Reference

This section contains all of the CORBA public interfaces accessible to a client, and may be used as a reference to services offered by our distributed proxy objects.

Four CORBA interfaces are defined within the module `disgraph` (this module name was introduced in order to resolve potential name-scoping issues). These interfaces are `proxy_interface`, `proxy_factory`, `EdgeIterator` and `NodeIterator`. A client implementation will need to refer only to these interfaces in order to access graph services and algorithms.

3.1 IDL for the Graph Proxy

"graph_proxy.idl" 4 ≡

```
module disgraph {
    ⟨graph_proxy.idl typedefs specific 5a⟩
    ⟨graph_proxy.idl typedefs edge 5b⟩
    ⟨graph_proxy.idl typedefs lists 5c⟩
    ⟨graph_proxy.idl typedefs representation 5d⟩
    ⟨graph_proxy.idl exceptions 6a⟩
    ⟨graph_proxy.idl iterators 6b⟩
    ⟨graph_proxy.idl interface 7a⟩
    ⟨graph_proxy.idl extract representation 7b⟩
    ⟨graph_proxy.idl node interface 8a⟩
    ⟨graph_proxy.idl edge interface 8b⟩
    ⟨graph_proxy.idl algorithm object calls 8c⟩
  };
  ⟨graph_proxy.idl proxy factory 9⟩
};
```

3.1.1 Data Types

Care has been taken to separate the data-type definitions from the rest of the code to facilitate future redefinition. We use CORBA built-in data types such as `CORBA::String`, `CORBA::Boolean`, and `CORBA::Sequence` to represent any data that is transmitted.

⟨`graph_proxy.idl` typedefs specific 5a⟩ ≡

```
typedef long NodeType;
typedef double Weight;
```

Used in part 4.

A graph edge is defined by a triplet $< node_i, node_j, weight >$, a compact representation of edges for the purposes of populating and retrieving data from the graph. An IDL `struct` is used for compound data types that are language- and platform-independent.

⟨`graph_proxy.idl` typedefs edge 5b⟩ ≡

```
struct EdgeType {
      NodeType        node_i;
      NodeType        node_j;
      Weight          weight;
};
```

Used in part 4.

CORBA sequences allow efficient transmission of blocks of data. Our timing tests (presented in a later section) show that `sequence` transmission is vastly more efficient than serial transmission of individual data items.

⟨`graph_proxy.idl` typedefs lists 5c⟩ ≡

```
typedef sequence <EdgeType> Edges;
typedef sequence <NodeType> Nodes;
```

Used in part 4.

Below, the interface's representation of a graph is defined as consisting of a `nodeList` and an `edgeList`. `stdGraphType` is used when transmitting data intended to represent all edges and nodes of a graph. This more abstract representation serves as a common representation, and is used by a client to specify an abstraction of a graph.

⟨`graph_proxy.idl` typedefs representation 5d⟩ ≡

```
struct stdGraphType {
      Nodes   nodeList;
      Edges   edgeList;
};
```

Used in part 4.

3.1.2 Exceptions

Below is the IDL definition of the types of exceptions that may be raised by the `graph_proxy`. However, since neither version 1.1.1 nor 1.1.2 of the Cygnus `egcs` compiler appear to fully support MICO exception-handling, our components do not make use of the below exceptions at this time.[1]

⟨graph_proxy.idl exceptions 6a⟩ ≡
```
exception InvalidNodeType { NodeType node; };
exception InvalidNodeList { Nodes nodeList; };
exception InvalidEdgeType { EdgeType edge; };
exception InvalidEdgeList { Edges edgeList; };
exception InvalidstdGraphType { stdGraphType graph; };

exception InvalidProxyInterface { };
exception BadAllocation { stdGraphType graph; };
exception BadFree { };
```

Used in part 4.

3.1.3 Iterators

Below, we introduce the concept of an iterator within the CORBA framework. Such a construct can be used to iterate though a context's binding when doing a nameservice lookup[Baker97].

⟨graph_proxy.idl iterators 6b⟩ ≡
```
interface NodeIterator {
  unsigned long next_one(out NodeType node);
  unsigned long next_n (in unsigned long how_many,
                out Nodes nl);
  unsigned long all ( out Nodes nl );
  unsigned long length ( );
  unsigned long reset ( );
  void destroy( );
};

interface EdgeIterator {
  unsigned long next_one(out EdgeType edge);
  unsigned long next_n (in unsigned long how_many,
                out Edges el);
  unsigned long all ( out Edges el );
  unsigned long length ( );
  unsigned long reset ( );
  void destroy( );
};
```

Used in part 4.

[1] Since our non-distributed graph package does not propagate exceptions, this is of little import.

3.1.4 Proxy Interface

Below is an IDL definition that provides a near one-to-one mapping to methods provided by `GraphInterface`. If additional algorithms/services were to be added to the `GraphInterface`, corresponding additions would have to be made to this IDL definition of the `graph_proxy`.

The interface permits fine-grain operations on a graph; such use may not be desirable due to the overhead incurred in the marshaling and demarshaling process.[2] A client will be responsible for determining the level of granularity most appropriate in its setting.

⟨`graph_proxy.idl` interface 7a⟩ ≡

```
interface proxy_interface {
 void destroy ( )
      raises (BadFree);
```

Used in part 4.

The allocation interface as defined by the IDL is not to be confused with any other reference to allocation within the graph data structures defined in Appendix A, which have to do with allocation of internal graph structures. These are not similar or even related. The calls below are offered to a client after a reference to `graph_proxy` has been obtained.

⟨`graph_proxy.idl` extract representation 7b⟩ ≡

```
stdGraphType getGraph ( )
      raises(InvalidstdGraphType);
```

Used in part 4.

The IDL provides several methods that interact with node input types for manipulation of a graph object:

[2] Such overhead may be significant—see the timing tests presented in a later section.

⟨graph_proxy.idl node interface 8a⟩ ≡
```
        Nodes adjacentNodesOut ( in NodeType node )
              raises (InvalidNodeType);
        Nodes adjacentNodesIn ( in NodeType node )
              raises (InvalidNodeType);
        boolean isConnected ( in NodeType node )
              raises (InvalidNodeType);
        Weight findWeight ( in NodeType node1, in NodeType node2 )
              raises (InvalidNodeType);
        Weight addWeights ( in Nodes nodeList )
              raises (InvalidNodeList);
        void removeOrphanNodes();
        boolean addNode ( in NodeType node )
              raises (InvalidNodeType);
        boolean addNodes ( in Nodes nodeList )
              raises (InvalidNodeList);
        boolean removeNodes ( in Nodes nodeList )
              raises (InvalidNodeList);
        Nodes getNodes ( );
        NodeIterator getINodes ( );
```

Used in part 4.

The following methods provide for manipulation of a graph's edges:

⟨graph_proxy.idl edge interface 8b⟩ ≡
```
        boolean addEdge ( in EdgeType edge )
              raises (InvalidEdgeType);
        boolean addEdges ( in Edges edgeList )
              raises (InvalidEdgeList);
        boolean removeEdge ( in NodeType node_i,
                             in NodeType node_j )
              raises (InvalidNodeType);
        Edges getEdges ( );
        EdgeIterator getIEdges ( );
```

Used in part 4.

Finally, proxy methods for calls to algorithms that act upon graphs are provided:

⟨graph_proxy.idl algorithm object calls 8c⟩ ≡
```
        boolean has_cycles ( );
        Nodes topological_sort( in NodeType u )
              raises (InvalidNodeType);
        proxy_interface prims ( in NodeType r )
              raises (InvalidNodeType);
```

Used in part 4.

3.2 IDL for the Proxy Factory

Next, we present the IDL definition for the `proxy_factory`, which handles the creation of `proxy_interfaces`.

When a client connects to a server, it should get a new CORBA object and not an object another client already is using. Without the functionality provided by this factory class all the clients connecting to the system would share the same object[Gamma95].

This construct also allows the client to create multiple `graph_interface` instances on the same server.

In addition to the option to `create` and `destroy` an interface, a client also is given the option to `clone` an interface.

⟨`graph_proxy.idl` proxy factory 9⟩ ≡

```
interface proxy_factory {
  proxy_interface createEmpty ( );
  proxy_interface create ( in stdGraphType graph )
      raises (InvalidstdGraphType, BadAllocation);
  proxy_interface createEdgeNode ( in Nodes nodeList,
                                   in Edges edgeList)
      raises (InvalidNodeList, InvalidEdgeList, BadAllocation);
  proxy_interface clone ( in proxy_interface gpi )
      raises (InvalidProxyInterface, BadAllocation);
};
```

Used in part 4.

4 Proxy Implementation Classes

4.1 Graph Proxy Implementation

The graph proxy's implementation class derives from `graph_proxy_skel`, which the IDL compiler generates from the IDL definition given in the previous section. This is responsible for implementing the methods and interfaces as they are defined by the `IDL`.

```
"graph_proxy_impl.h" 10a ≡
    #ifndef __GRAPH_PROXY_IMPL_H__
    #define __GRAPH_PROXY_IMPL_H__

    ⟨graph_proxy_impl header includes 10b⟩
    ⟨graph_proxy_impl header class 11a⟩
    ⟨graph_proxy_impl node iterator class 11b⟩
    ⟨graph_proxy_impl edge iterator class 14a⟩

    class proxy_interface_impl :
          virtual public proxy_interface_skel
    {
      ⟨graph_proxy_impl interface private methods 16⟩
      ⟨graph_proxy_impl interface methods structors 17a⟩
      ⟨graph_proxy_impl interface methods extract representation 17b⟩
      ⟨graph_proxy_impl interface methods sequence return type 18a⟩
      ⟨graph_proxy_impl interface methods boolean return type 18b⟩
      ⟨graph_proxy_impl interface methods scalar return type 19⟩
      ⟨graph_proxy_impl interface methods node manipulator 20⟩
      ⟨graph_proxy_impl interface methods node retrieval 21a⟩
      ⟨graph_proxy_impl interface methods edge manipulator 21b⟩
      ⟨Has Cycles implemented 23b⟩
      ⟨Topological sort implemented 24⟩
      ⟨Prims algorithm implemented 25⟩
    };
    ⟨graph_proxy_impl factory class 26a⟩

    #endif
```

The `graph_proxy_impl` class provides the implementation of the proxy components defined in the IDL. Its methods map STL types to CORBA-defined types and vice-versa, implementing a full set of proxy components.

```
⟨graph_proxy_impl header includes 10b⟩ ≡
    #include "graph_proxy.h"
    #include "matrixgraph.h"
    #include "adjacencygraph.h"
    #include "algorithms.h"
```

Used in part 10a.

Below, the non-distributed graph type handled by this server is determined by a preprocessor definition passed in as a command-line option.

⟨graph_proxy_impl header class 11a⟩ ≡
```
using namespace disgraph;

#ifdef MAT
typedef matrix_graph<NodeType, NodeType> GraphType;
#else
typedef adjacency_graph<NodeType, NodeType> GraphType;
#endif
```

Used in part 10a.

4.1.1 Iterators

Below, we implement two types of iterators to allow the user to iterate through return types. These are not iterators as defined by STL, but are useful for incremental extraction of data in a distributed setting[Baker97]. A client's memory may be limited; such iteration will enable a client to use minimal memory for examining returned data.

4.1.2 Node Iteration

The `NodeIterator_impl` class provides for incremental retrieval of node data.

⟨graph_proxy_impl node iterator class 11b⟩ ≡
```
class NodeIterator_impl :
        virtual public NodeIterator_skel {
private:
  CORBA::ULong idx;
  Nodes nlist;

public:
  NodeIterator_impl( const Nodes& nl ): idx(0), nlist(nl) {}
  virtual ~NodeIterator_impl ( ) {
        nlist.length(0);
  };
  ⟨node next_one 12a⟩
  ⟨node next_n 12b⟩
  ⟨node all 12c⟩
  ⟨node length 13a⟩
  ⟨node reset 13b⟩
  ⟨node destroy 13c⟩
};
```

Used in part 10a.

The below method, `next_one`, retrieves the next node and advances the iterator.

⟨node next_one 12a⟩ ≡

```
CORBA::ULong
NodeIterator_impl::next_one ( NodeType& node ) {
  if (idx < nlist.length() ) {
    node = nlist[idx++];
    return 1;
  } else return 0;
}
```

Used in part 11b.

The next_n method retrieves a passed-in number of nodes, and advances the iterator accordingly.

⟨node next_n 12b⟩ ≡

```
CORBA::ULong
NodeIterator_impl::next_n( CORBA::ULong how_many,
                           Nodes*& nl ) {

  CORBA::Long left = nlist.length() - idx;
  CORBA::ULong num_ret =
      ( ( left >= how_many ) ? how_many : left );
  nl = new Nodes(num_ret);
  nl->length (num_ret);
  for (CORBA::ULong i = 0; i < num_ret; ++i, ++idx)
    (*nl)[i] = nlist[idx];
  return num_ret;
}
```

Used in part 11b.

The all method returns all nodes in the graph, regardless of the current iterator position. The iterator position remains unchanged.

⟨node all 12c⟩ ≡

```
CORBA::ULong
NodeIterator_impl::all ( Nodes*& nl ) {
    CORBA::ULong num_ret = nlist.length();
    nl = new Nodes(num_ret);
    nl->length(num_ret);
    for (CORBA::ULong i = 0; i < num_ret; ++i)
       (*nl)[i] = nlist[i];
    return num_ret;
 }
```

Used in part 11b.

The below method, length, yields the number of nodes in the graph.

⟨node **length** 13a⟩ ≡

```
CORBA::ULong NodeIterator_impl::length ( ) {
        return nlist.length();
 }
```

Used in part 11b.

`reset` returns the iterator position to the beginning of the node list:

⟨node **reset** 13b⟩ ≡

```
CORBA::ULong NodeIterator_impl::reset ( ) {
        return (idx = 0);
 }
```

Used in part 11b.

Finally, `destroy` deletes the iterator and reclaims its memory.

⟨node **destroy** 13c⟩ ≡

```
void NodeIterator_impl::destroy ( ) {
        delete this;
 }
```

Used in part 11b.

4.1.3 Edge Iteration

Below is an implementation of an Edge Iterator for incremental examination of a graph's edge data. The functionality of each of its proxy methods is analogous to that of the corresponding `NodeIterator` methods.

⟨graph_proxy_impl edge iterator class 14a⟩ ≡

```
class EdgeIterator_impl :
      virtual public EdgeIterator_skel {
private:
  CORBA::ULong idx;
  Edges elist;

public:
  EdgeIterator_impl( const Edges& el )
        : idx(0), elist(el) { }
  virtual ~EdgeIterator_impl ( ) {
        elist.length(0);
  };
  ⟨edge next_one 14b⟩
  ⟨edge next_n 14c⟩
  ⟨edge all 15a⟩
  ⟨edge length 15b⟩
  ⟨edge reset 15c⟩
  ⟨edge destroy 15d⟩
};
```

Used in part 10a.

⟨edge next_one 14b⟩ ≡

```
CORBA::ULong
EdgeIterator_impl::next_one ( EdgeType& edge ) {
   if (idx < elist.length() ) {
      edge = elist[idx++];
      return 1;
   } else return 0;
}
```

Used in part 14a.

⟨edge next_n 14c⟩ ≡

```
CORBA::ULong
EdgeIterator_impl::next_n( CORBA::ULong how_many,
                           Edges*& el ) {
  CORBA::Long left = elist.length() - idx;
  CORBA::ULong num_ret =
      ( ( left >= how_many ) ? how_many : left );
  el = new Edges(num_ret);
  el->length (num_ret);
  for (CORBA::ULong i = 0; i < num_ret; ++i, ++idx)
    (*el)[i] = elist[idx];
  return num_ret;
}
```

Used in part 14a.

⟨edge `all` 15a⟩ ≡

```
CORBA::ULong
EdgeIterator_impl::all ( Edges*& el ) {
    CORBA::ULong num_ret = elist.length();
    el = new Edges(num_ret);
    el->length(num_ret);
    for (CORBA::ULong i = 0; i < num_ret; ++i)
       (*el)[i] = elist[i];
    return num_ret;
 }
```

Used in part 14a.

⟨edge `length` 15b⟩ ≡

```
CORBA::ULong
 EdgeIterator_impl::length ( ) {
       return elist.length();
 }
```

Used in part 14a.

⟨edge `reset` 15c⟩ ≡

```
CORBA::ULong
 EdgeIterator_impl::reset ( ) {
       return (idx = 0);
 }
```

Used in part 14a.

⟨edge `destroy` 15d⟩ ≡

```
void
 EdgeIterator_impl::destroy ( ) {
       delete this;
 }
```

Used in part 14a.

4.1.4 Interface Methods

A client of this server should not be constrained to making calls to a particular graph representation, as long as the graph's public interface is what the client expects.

In our system, a server can be created specifically for each type of graph representation. When a client contacts the naming service, that client can request a particular representation.

The server is responsible for the construction of a `GraphInterface` object. On the other hand, the client is responsible for the selection of the appropriate internal representation and population of the representation, as well as its eventual deletion.

⟨graph_proxy_impl interface private methods 16⟩ ≡

```
    private:
      GraphType* dg;

      has_cycle<NodeType, NodeType>* cycle;
      topo_sort<NodeType, NodeType>* topsort;
      mst_prim<NodeType, NodeType>* mst;
```

Used in part 10a.

We now present the proxy methods. One major concern is the error checking that the proxy must do to appropriately propagate an error over the CORBA connection. To address this concern, a standard set of exceptions has been defined in the IDL specification of the `graph_proxy` interface. However, due to lack of `egcs` compiler support (`MICO` exceptions often are never actually thrown) at the time of this writing, these were not implemented.

The client is responsible for maintaining any mapping between node numbers in the distributed graph and the original node data and labels (which may be of any type). We will, in a later section, present an interactive client that does exactly this.

4.1.5 Construction and Destruction

The constructor method for the `graph_proxy_impl` class contains no operations.

However, the deconstructor method should make sure that allocated memory, if any, is reclaimed from the graph. However, a client is responsible for making sure that this destruction is called explicitly when the memory is no longer required. If a client exits without calling the appropriate `destroy` method, the server does not reclaim memory. `Arno Puder`, co-creator of `MICO`, in an email discussion, confirmed that such explicit destruction calls *must* be made: "(The destructor) will not be called implicitly when there is no more reference pointing to the object. CORBA does not do a global garbage collection. However, if you dispose the object explicitly, then of course the destructor will be called."[3]

For this purpose, `graph_proxy_impl` provides a `destroy` method a client must call explicitly for destruction of a server object. As Bjarne Stroustrup, creator of C++, has stated, "Explicit calls of destructors ... should be avoided whenever possible. Occasionally, they are essential."[Stroustrup97] We have here a case where the use of explicit destruction is unavoidable.

[3]Notice that this lack of garbage collection means that if a client crashes or for other unexpected reasons fails to destroy an object, significant server-side memory leaks may occur.

⟨graph_proxy_impl interface methods structors 17a⟩ ≡
```
       public:
         proxy_interface_impl ( ) {
           dg = new GraphType( true );

           cycle = new has_cycle<NodeType, NodeType>();
           topsort = new topo_sort<NodeType, NodeType>();
           mst = new mst_prim<NodeType, NodeType>();
         }

         virtual ~proxy_interface_impl ( ) {
             if (dg != NULL) delete( dg );
             if (cycle != NULL) delete( cycle );
             if (topsort != NULL) delete( topsort );
             if (mst != NULL) delete( mst );
         }

         void destroy ( )
           throw(::CORBA::SystemException_var,
             BadFree_var)
         { delete this; }
```

Used in part 10a.

4.1.6 Data Extraction and Manipulation

⟨graph_proxy_impl interface methods extract representation 17b⟩ ≡
```
         stdGraphType* getGraph()
           throw(
             ::CORBA::SystemException_var,
             InvalidstdGraphType_var)
          {
             stdGraphType* retval = new stdGraphType;
             Nodes* nl = getNodes();
             Edges* el = getEdges();
             retval->edgeList = *el;
             retval->nodeList = *nl;
             delete nl;  // need to clean up
             delete el;  // need to clean up
             return retval;
          }
```

Used in part 10a.

In choosing algorithms and services to implement, we took care to choose ones such that many different styles of proxies could be demonstrated.

The below methods demonstrate the conversion to a `CORBA::Sequence` from a `vector` return. Performing this conversion in a single process space is more efficient than using an iterator over the network[Baker97]; to minimize network traffic the entire `CORBA::Sequence` is returned.

⟨`graph_proxy_impl` interface methods sequence return type 18a⟩ ≡

```
         Nodes* adjacentNodesOut( NodeType node )
           throw(::CORBA::SystemException_var,
             InvalidNodeType_var)
           {
             vector<NodeType> nl;
             dg->adjacentNodesOut(node, back_inserter(nl));

             Nodes* retval = new Nodes(nl.size());
             retval->length(nl.size());
             for (CORBA::ULong i = 0; i < nl.size(); ++i)
                 (*retval)[i] = nl[i];
             return retval;
           }

         Nodes* adjacentNodesIn( NodeType node )
           throw(::CORBA::SystemException_var,
             InvalidNodeType_var)
           {
             vector<NodeType> nl;
             dg->adjacentNodesIn(node, back_inserter(nl));

             Nodes* retval = new Nodes(nl.size());
             retval->length(nl.size());
             for (CORBA::ULong i = 0; i < nl.size(); ++i)
               (*retval)[i] = nl[i];
             return retval;
          }
```

Used in part 10a.

The below method demonstrates the `Boolean` return capabilities of CORBA and the simplicity of creating a proxy methods once the framework is in place. Calls from client to server are simply passed on to the `GraphInterface`; since this occurs in a single process space, very little overhead is incurred.

⟨`graph_proxy_impl` interface methods boolean return type 18b⟩ ≡

```
        CORBA::Boolean isConnected( NodeType node )
          throw (::CORBA::SystemException_var,
            InvalidNodeType_var)
         {
             return dg->isConnected ( node );
         }
```

Used in part 10a.

The below proxy method demonstrates the return of a scalar value.

⟨`graph_proxy_impl` interface methods scalar return type 19⟩ ≡

```
Weight addWeights( const Nodes& nodeList )
  throw(::CORBA::SystemException_var,
    InvalidNodeType_var)
 {
    vector<NodeType> conv(nodeList.length());
    for (CORBA::ULong i = 0; i < nodeList.length(); ++i)
      conv[i] = nodeList[i];
    return dg->addWeights(conv.begin(), conv.end());
 }
```

Used in part 10a.

The below node-manipulation methods convert the `in` parameter types (if any) to types acceptable to the `graph_interface` package. The newly-constructed object, if so needed, is then passed on.

⟨graph_proxy_impl interface methods node manipulator 20⟩ ≡

```
void removeOrphanNodes()
{
    dg->removeOrphanNodes();
}

CORBA::Boolean addNode( NodeType node )
  throw(::CORBA::SystemException_var,
    InvalidNodeType_var)
{
    dg->addNode( node );
    return true;
}

CORBA::Boolean addNodes( const Nodes& nodeList )
  throw(::CORBA::SystemException_var,
    InvalidNodeList_var)
{
    for (CORBA::ULong i = 0; i < nodeList.length(); ++i)
      dg->addNode( nodeList[i] );
    return true;
}

CORBA::Boolean removeNodes( const Nodes& nodeList )
  throw(::CORBA::SystemException_var,
    InvalidNodeList_var)
{
    vector<NodeType> conv(nodeList.length());
    for (CORBA::ULong i = 0; i < nodeList.length(); ++i)
      conv[i] = nodeList[i];
    dg->removeNodes(conv.begin(), conv.end());
    return true;
}
```

Used in part 10a.

⟨graph_proxy_impl interface methods node retrieval 21a⟩ ≡

```
Nodes* getNodes() {
    vector<NodeType> nl;
    copy(dg->begin(), dg->end(), back_inserter(nl));
    Nodes* retval = new Nodes(nl.size());
    retval->length(nl.size());
    for (CORBA::ULong i = 0; i < nl.size(); ++i)
      (*retval)[i] = nl[i];
    return retval;
}

NodeIterator_ptr getINodes() {
    vector<NodeType> nl;
    copy(dg->begin(), dg->end(), back_inserter(nl));
    Nodes retval;
    retval.length(nl.size());
    for (CORBA::ULong i = 0; i < nl.size(); ++i)
      retval[i] = nl[i];
    NodeIterator_ptr ni = new NodeIterator_impl( retval );
    return NodeIterator::_duplicate (ni);
}
```

Used in part 10a.

By now, a well-established pattern of extraction and conversion has been established for the creation of proxy methods. The below methods use this paradigm to extract edge information:

⟨graph_proxy_impl interface methods edge manipulator 21b⟩ ≡

⟨gpi Adding and removing edges 23a⟩
⟨gpi findWeight 21c⟩
⟨gpi getEdges 22a⟩
⟨gpi getIEdges 22b⟩

Used in part 10a.

⟨gpi findWeight 21c⟩ ≡

```
Weight findWeight ( NodeType node1, NodeType node2 )
  throw(
    ::CORBA::SystemException_var,
    InvalidNodeList_var)
{
    return dg->findWeight( node1, node2 );
}
```

Used in part 21b.

⟨gpi getEdges 22a⟩ ≡

```
        Edges* getEdges() {
            typedef pair<NodeType, NodeType> OneEdge;
            vector<OneEdge> severalEdges;

            dg->getEdges( back_inserter( severalEdges ) );

            vector<Weight> theWeights;
            for( CORBA::ULong i = 0; i < severalEdges.size(); ++i )
              theWeights.push_back( dg->findWeight(
                severalEdges[i].first, severalEdges[i].second ) );

            Edges* retval = new Edges(severalEdges.size());
            retval->length(severalEdges.size());
            for( CORBA::ULong i = 0; i < severalEdges.size(); ++i) {
               ((*retval)[i]).node_i = severalEdges[i].first;
               ((*retval)[i]).node_j = severalEdges[i].second;
               ((*retval)[i]).weight = theWeights[i];
            }
            return retval;
         }
```

Used in part 21b.

⟨gpi getIEdges 22b⟩ ≡

```
        EdgeIterator_ptr getIEdges() {
            typedef pair<NodeType, NodeType> OneEdge;
            vector<OneEdge> severalEdges;

            dg->getEdges( back_inserter( severalEdges ) );

            vector<Weight> theWeights;
            for( CORBA::ULong i = 0; i < severalEdges.size(); ++i )
              theWeights.push_back( dg->findWeight(
                severalEdges[i].first, severalEdges[i].second ) );

            Edges retval;
            retval.length(severalEdges.size());
            for( CORBA::ULong i = 0; i < severalEdges.size(); ++i) {
               retval[i].node_i = severalEdges[i].first;
               retval[i].node_j = severalEdges[i].second;
               retval[i].weight = theWeights[i];
            }

            EdgeIterator_ptr ei = new EdgeIterator_impl( retval );
            return EdgeIterator::_duplicate (ei);
        }
```

Used in part 21b.

The below methods provide for addition and removal of edges:

⟨gpi Adding and removing edges 23a⟩ ≡

```
CORBA::Boolean addEdge( const EdgeType& edge )
  throw(::CORBA::SystemException_var,
    InvalidEdgeType_var)
{
    return dg->addEdge(edge.node_i, edge.node_j,
                       edge.weight);
}

CORBA::Boolean addEdges( const Edges& edgeList )
  throw(::CORBA::SystemException_var,
    InvalidEdgeList_var)
{
    for (CORBA::ULong i = 0; i < edgeList.length();++i)
      if( !addEdge(edgeList[i]) )
        return false;
    return true;
}

CORBA::Boolean
removeEdge( NodeType node_i, NodeType node_j)
  throw(::CORBA::SystemException_var,
    disgraph::InvalidNodeType_var)
{
    return dg->removeEdge(node_i, node_j);
}
```

Used in part 21b.

4.1.7 Algorithm Calls

The below algorithm calls each invoke the `run` method of the appropriate algorithm class (in `algorithms.h`; see Appendix B).

⟨Has Cycles implemented 23b⟩ ≡

```
CORBA::Boolean has_cycles ( ) {
    return cycle->run( *dg );
}
```

Used in part 10a.

The `topological_sort` algorithm returns a sequence of nodes; the below proxy method converts the return from `run` into a form suitable for transmission.

⟨Topological sort implemented 24⟩ ≡

```
Nodes* topological_sort( NodeType u )
throw(
  ::CORBA::SystemException_var,
  InvalidNodeType_var)
{
  vector<NodeType> nodesBack;
  topsort->run( *dg, u, back_inserter( nodesBack ) );

  Nodes* retval = new Nodes(nodesBack.size());
  retval->length( nodesBack.size() );
  for( CORBA::ULong i = 0; i < nodesBack.size(); ++i )
    (*retval)[i] = nodesBack[i];

  return retval;
}
```

Used in part 10a.

Our implementation of Prim's minimum spanning tree algorithm returns a handle to a `proxy_interface` object. The below proxy method therefore must do a bit more work than the previous two in order to convert the data returned from `run` into a form suitable for CORBA transport.

⟨Prims algorithm implemented 25⟩ ≡

```
virtual proxy_interface_ptr prims ( NodeType r )
 throw( ::CORBA::SystemException_var,
  InvalidNodeType_var)
{
  GraphType returnGraph(true);
  returnGraph = mst->run( *dg, r );

  vector<NodeType> nodesBack;
  copy( returnGraph.begin(), returnGraph.end(),
        back_inserter( nodesBack ) );

  Nodes* returnNodes = new Nodes(nodesBack.size());
  returnNodes->length(nodesBack.size());
  for( CORBA::ULong i = 0; i < nodesBack.size(); ++i)
    (*returnNodes)[i] = nodesBack[i];

  typedef pair<NodeType, NodeType> OneEdge;
  vector<OneEdge> severalEdges;
  returnGraph.getEdges( back_inserter( severalEdges ) );

  vector<Weight> theWeights;
  for( CORBA::ULong i = 0; i < severalEdges.size(); ++i )
    theWeights.push_back(
      returnGraph.findWeight( severalEdges[i].first,
                              severalEdges[i].second ) );

  vector<EdgeType> edgesBack;
  Edges* returnEdges = new Edges(severalEdges.size());
  returnEdges->length(severalEdges.size());
  for( CORBA::ULong i = 0; i < severalEdges.size(); ++i) {
    ((*returnEdges)[i]).node_i = severalEdges[i].first;
    ((*returnEdges)[i]).node_j = severalEdges[i].second;
    ((*returnEdges)[i]).weight = theWeights[i];
  }

  proxy_interface_ptr ngpi = new proxy_interface_impl;

  ngpi->addNodes( *returnNodes );
  ngpi->addEdges( *returnEdges );

  delete returnNodes;  //need to clean up
  delete returnEdges;  //need to clean up

  return proxy_interface::_duplicate( ngpi );
}
```

Used in part 10a.

4.2 Proxy Factory Implementation

The proxy factory is responsible for the creation of `graph_interface` objects.

⟨`graph_proxy_impl` factory class 26a⟩ ≡

```
class proxy_factory_impl :
        virtual public proxy_factory_skel
{
 public:
  proxy_factory_impl ( ) {}
  virtual ~proxy_factory_impl ( ) {}

  ⟨factory create 26b⟩
  ⟨factory createEdgeNode 27a⟩
  ⟨factory createEmpty 27b⟩
  ⟨factory clone 28a⟩
};
```

Used in part 10a.

Given an abstract representation of a graph (`stdGraphType`), a client can create a proxy interface.

⟨factory `create` 26b⟩ ≡

```
  virtual proxy_interface_ptr
    create( const stdGraphType& graph )
      throw(
        ::CORBA::SystemException_var,
        InvalidstdGraphType_var,
        BadAllocation_var)
 {
    proxy_interface_ptr gpi  =
        new proxy_interface_impl;
    gpi->addNodes( graph.nodeList );
    gpi->addEdges( graph.edgeList );
    return proxy_interface::_duplicate (gpi);
  }
```

Used in part 26a.

A client may also create a proxy interface by passing a list of nodes and a list of edges.

⟨factory `createEdgeNode` 27a⟩ ≡

```
virtual proxy_interface_ptr
  createEdgeNode( const Nodes& nodeList,
                  const Edges& edgeList )
throw( ::CORBA::SystemException_var,
  InvalidNodeList_var,
  InvalidEdgeList_var,
  BadAllocation_var)
{
  proxy_interface_ptr gpi  =
      new proxy_interface_impl;
  gpi->addNodes( nodeList );
  gpi->addEdges( edgeList );
  return proxy_interface::_duplicate (gpi);
}
```

Used in part 26a.

An empty proxy interface may also be requested:

⟨factory `createEmpty` 27b⟩ ≡

```
virtual proxy_interface_ptr createEmpty ( ) {
  proxy_interface_ptr gpi  =
      new proxy_interface_impl;
  return proxy_interface::_duplicate (gpi);
}
```

Used in part 26a.

The `clone` method allows a client to copy an existing interface:

⟨factory `clone` 28a⟩ ≡

```
virtual proxy_interface_ptr clone
  ( proxy_interface_ptr gpi)
  throw( ::CORBA::SystemException_var,
    InvalidProxyInterface_var,
    BadAllocation_var)
{
  proxy_interface_ptr ngpi  =
      new proxy_interface_impl;
  Nodes * nl = gpi->getNodes();
  Edges * el = gpi->getEdges();

  ngpi->addNodes( *nl );
  ngpi->addEdges( *el );

  delete nl;  //need to clean up
  delete el;  //need to clean up
  return proxy_interface::_duplicate (ngpi);
}
```

Used in part 26a.

5 Example Servers and Clients

This section presents code for the `server` and `client`. Currently, the client constructs both static (SII) calls as well as DII calls. Although computationally costly, DII calls are more flexible than SII calls.

5.1 Server

`"server_ns.cc"` 28b ≡

```
⟨server_ns includes 29a⟩
int main( int argc, char *argv[] ) {
  ⟨server_ns orb setup 29b⟩
  ⟨server_ns nameservices 29c⟩
  ⟨server_ns run orb 31b⟩
}
```

This is a simple server but it is robust enough to demonstrate the major features of the component implementation. An end user will decide on which mode this server will run. In our examples, the server is run in shared mode[Baker97].

⟨server_ns includes 29a⟩ ≡

```
#include <CORBA.h>
#include <mico/naming.h>
#include <iostream.h>
#include "graph_proxy.h"
#include "graph_proxy_impl.h"

using namespace disgraph;
```

Used in part 28b.

The below code is used to set up the MICO ORB. Then, a proxy_factory is created—a client can use this to create a proxy_interface for access to the non-distributed graph classes (Appendix A).

⟨server_ns orb setup 29b⟩ ≡

```
CORBA::ORB_var orb =
      CORBA::ORB_init( argc, argv, "mico-local-orb" );
CORBA::BOA_var boa =
      orb->BOA_init (argc, argv, "mico-local-boa");

cerr << "[Initialization of the server complete]" << endl;

proxy_factory_ptr
      myGraphProxyFactory = new proxy_factory_impl();
```

Used in part 28b.

Below, the server object contacts the naming services and registers itself.

⟨server_ns nameservices 29c⟩ ≡

⟨sns contact naming service 30⟩
⟨sns register with naming service 31a⟩

Used in part 28b.

⟨sns contact naming service 30⟩ ≡

```
CosNaming::NamingContext_var nc;
try {
  CORBA::Object_var nsobj =
     orb->resolve_initial_references ("NameService");

  nc = CosNaming::NamingContext::_narrow (nsobj);
  if ( CORBA::is_nil (nc) ) {
    cerr << "Failed to narrow naming context" << endl;
    exit(-1);
  }
}
catch(CORBA::ORB::InvalidName& ex) {
  cerr << "Service required is invalid [does not exist]."
       << endl;
  exit(-1);
}
catch(...) {
  cerr << "Unknown exception thrown at resolving" << endl;
  exit(-1);
}
```

Used in part 29c.

Below, the server object registers with the naming service. When the `bind` fails (this will occur if the requested name is already bound to another object), a call to `rebind` causes the naming service to register the current object, overriding any existing references.

⟨sns register with naming service 31a⟩ ≡

```
        CosNaming::Name name;
        try {
          name.length (1);
          name[0].id = CORBA::string_dup ("graph_proxy_factory");

#ifdef MAT
          name[0].kind = CORBA::string_dup ("matrix");
          cerr << "[Matrix Server Registering]" << endl;
#else
          name[0].kind = CORBA::string_dup ("adjacency");
          cerr << "[Adjacency Server Registering]" << endl;
#endif

          try {
            nc->bind (name, myGraphProxyFactory);
          }
          catch(CosNaming::NamingContext::AlreadyBound& ex) {
            nc->rebind (name, myGraphProxyFactory);
          }
          catch(...) {
            nc->rebind (name, myGraphProxyFactory);
          }
          cerr << "[Server Name Registered]" << endl;
        }
        catch (CORBA::COMM_FAILURE& ex) {
          cerr << "Caught system exception COMM_FAILURE, unable to"
             << " contact the naming service." << endl;
          exit(-1);
        }
        catch (...) {
          cerr << "Caught unknown exception while"
             << " contacting the naming service." << endl;
          exit(-1);
        }
```

Used in part 29c.

Once the naming service has been notified, the server is ready to begin serving the `proxy_factory` instance to the various clients.

⟨`server_ns` run orb 31b⟩ ≡

```
        boa->impl_is_ready ( CORBA::ImplementationDef::_nil( ));
        orb->run ();
        return 0;
```

Used in part 28b.

5.2 Client

The sample client presented in this subsection demonstrates a wide range of the functionality offered by our system.

```
"client_ns.cc" 32a ≡
      ⟨client includes 32b⟩
      ⟨client helper functions 32c⟩
      int main ( int argc, char * argv[]) {
        ⟨client orb setup 33d⟩
        ⟨client nameservices 34a⟩
        ⟨client SII call allocation 36⟩
        ⟨client SII call first run 37a⟩
        ⟨client SII call algorithms 37b⟩
        ⟨client SII call iterator 40a⟩
        ⟨client SII call second series 42b⟩
        ⟨client SII call edge stuff 40b⟩
        ⟨client DII call node list return 43⟩
        ⟨client DII call scalar return 44⟩
      }
```

```
⟨client includes 32b⟩ ≡
      #include <CORBA.h>
      #include "graph_proxy.h"
      #include <iostream.h>
      #include <mico/naming.h>

      using namespace disgraph;
```

Used in parts 32a, 45a, 48a, 62.

A handful of functions are provided to simplify interface calls:

```
⟨client helper functions 32c⟩ ≡
      ⟨putNode 32d⟩
      ⟨putEdge 33a⟩
      ⟨printEdges 33b⟩
      ⟨printNodes 33c⟩
```

Used in parts 32a, 45a, 48a, 61.

```
⟨putNode 32d⟩ ≡
      NodeType putNode( NodeType x ) {
        return x;
      }
```

Used in part 32c.

⟨putEdge 33a⟩ ≡

```
EdgeType putEdge( NodeType x, NodeType y,
              Weight w ) {
  EdgeType anEdge;
  anEdge.node_i = x;
  anEdge.node_j = y;
  anEdge.weight = w;
  return anEdge;
}
```

Used in part 32c.

⟨printEdges 33b⟩ ≡

```
void printEdges( Edges * edgesBack ) {
  cout << "EDGE LISTING :" << endl;
  for( int i = 0; i < (*edgesBack).length(); ++i ) {
    cout << ((*edgesBack)[i]).node_i << " ";
    cout << ((*edgesBack)[i]).node_j << " ";
    cout << ((*edgesBack)[i]).weight << endl;
  }
}
```

Used in part 32c.

⟨printNodes 33c⟩ ≡

```
void printNodes( Nodes * nodesBack ) {
  cout << "NODE LISTING :" << endl;
  for( int i = 0; i < (*nodesBack).length(); ++i )
    cout << (*nodesBack)[i] << endl;
}
```

Used in part 32c.

Below, the `MICO` ORB is initialized.

⟨client orb setup 33d⟩ ≡

```
  Nodes nodes_list;
  NodeType node_num;

  CORBA::ORB_var orb =
        CORBA::ORB_init( argc, argv, "mico-local-orb" );
```

Used in parts 32a, 45a, 48a, 61.

Below, the client gets its naming context, and then requests a reference to a `proxy_factory` instance created by the server.

⟨`client` nameservices 34a⟩ ≡
    ⟨Client get naming context 34b⟩
    ⟨Client request proxy factory 35⟩

Used in parts 32a, 45a, 48a, 61.

⟨Client get naming context 34b⟩ ≡

```
CosNaming::NamingContext_var nc;
try {
  CORBA::Object_var nsobj =
    orb->resolve_initial_references ("NameService");

  nc = CosNaming::NamingContext::_narrow (nsobj);
  if ( CORBA::is_nil (nc) ) {
    cerr << "Failed to narrow naming context" << endl;
    exit(-1);
  }
}

catch(CORBA::ORB::InvalidName& ex) {
  cerr << "Service required is invalid [does not exist]."
       << endl;
  exit(-1);
}

catch(...) {
  cerr << "Unknown exception thrown at resolving" << endl;
  exit(-1);
}
```

Used in part 34a.

Below, the client chooses, at run-time, the type of graph server it will access:

⟨Client request proxy factory 35⟩ ≡

```
CosNaming::Name name;
CORBA::Object_var obj;
proxy_factory_var fdg;
int input;

try {
  name.length (1);
  name[0].id = CORBA::string_dup ("graph_proxy_factory");
  cout << "What type of g server would you like (0=adj/1=mat)? ";
  cin >> input;
  if (input == 0)
     name[0].kind = CORBA::string_dup ("adjacency");
  else
     name[0].kind = CORBA::string_dup ("matrix");

  obj = nc->resolve (name);
  fdg = proxy_factory::_narrow( obj );
}
catch (CORBA::COMM_FAILURE& ex) {
   cerr << "Caught system exception COMM_FAILURE, unable to"
        << " contact the naming service." << endl;
   exit(-1);
}
catch (...) {
   cerr << "Caught unknown exception while"
       << " contacting the naming service." << endl;
   exit(-1);
}
```

Used in part 34a.

Note that the type `proxy_interface_var` is required rather than the type `proxy_interface_ptr`. The distinction between the two is quite simple, but invariably, when a mistake is made, client results are somewhat unpredictable.

CORBA defines two types of variables that are available to the programmer. The first, `_ptr`, is probably the most flexible. When this one is used it is the responsibility of the programmer to manipulate the reference count and call `CORBA::_release` when done using it. The second, `_var`, automatically takes care of the manipulation of the reference count as to not burden the user. It is highly recommended, when writing a client, to use `_var` unless other flexibility is needed. The omission of proper manipulation of the count causes unpredictable results and may cause the system to fail. Use of `CORBA::_release` on a `_var` will produce incorrect results.

In the below code, the user is prompted for the number of nodes in a graph to be allocated. Such a graph is then allocated using SII.

⟨client SII call allocation 36⟩ ≡

```
int size;
cout << "Enter the size: ";
cin >> size;

proxy_interface_var dg;

cout << "Calling init";
cout << "Testing allocation" << endl;
stdGraphType* graph = new stdGraphType;
(graph->nodeList).length(size);
for (int j = 1; j <= size; j++)
  (graph->nodeList)[j-1] = j;

(graph->edgeList).length(size);
for (int count = 2; count <= size; count++) {
        (graph->edgeList)[count-2].node_i = 1;
        (graph->edgeList)[count-2].node_j = count;
        (graph->edgeList)[count-2].weight = 3000;
}

dg = fdg->create( (*graph) );
cout << "Allocated Graph" << endl;
```

Used in part 32a.

Below, a variety of SII calls are made.

⟨client SII call first run 37a⟩ ≡

```
Nodes* nodeList = dg->adjacentNodesOut( 1 );
cout << "Call Adjacent Nodes Out complete" << endl;
cout << "Printing out return: " << nodeList->length() << endl;
printNodes( nodeList );

cout << "Trying getNodes..." << endl;
nodeList = dg->getNodes();
cout << "Call Get Nodes complete" << endl;
cout << "Printing out return: " << nodeList->length() << endl;
printNodes( nodeList );

node_num = 1;
nodes_list.length(1);
nodes_list[0] = 4;
dg->removeNodes( nodes_list );
cout << "Call Remove Nodes complete" << endl;

cout << "Running Adjacent Nodes Out again" << endl;
nodeList = dg->adjacentNodesOut( node_num );
cout << "Call Adjacent Nodes Out complete" << endl;
cout << "Printing out return: " << nodeList->length() << endl;
printNodes( nodeList );

nodeList = dg->adjacentNodesIn( node_num );
cout << "Call Adjacent Nodes In complete" << endl;
cout << "Printing out return: " << nodeList->length() << endl;
printNodes( nodeList );
```

Used in part 32a.

⟨client SII call algorithms 37b⟩ ≡

⟨has_cycles test 38a⟩
⟨topological_sort test 38b⟩
⟨min_st test 39⟩

Used in part 32a.

Below, a pair of calls to `has_cycles` are made on a very simple graph. The first call should produce false; with the addition of an edge to the graph, a cycle is created—this is reflected in the return from the second call to `has_cycles`.

⟨has_cycles test 38a⟩ ≡

```
cout << "Calling has_cycle (should be false)..." << endl;
proxy_interface_var dg2 = fdg->createEmpty();

dg2->addNode ( putNode( 0 ));
dg2->addNode ( putNode( 1 ));
dg2->addNode ( putNode( 2 ));

dg2->addEdge ( putEdge( 0, 1, 0 ) );
dg2->addEdge ( putEdge( 1, 2, 0 ) );

if( dg2->has_cycles() )
cout << "Has a cycle";
else cout << "No cycle!";
cout << endl;

dg2->addEdge ( putEdge( 2, 0, 0 ) );

if( dg2->has_cycles() )
  cout << "Has a cycle";
else cout << "No cycle!";
cout << endl;

printEdges( dg2->getEdges() );
```

Used in part 37b.

Below, a simple three-layer graph undergoes a topological sort:

⟨topological_sort test 38b⟩ ≡

```
dg2->removeEdge( 1, 2 );
dg2->removeEdge( 2, 0 ); // remove cycle

dg2->addNode ( putNode( 3 ) );
dg2->addEdge ( putEdge( 0, 2, 0 ) );
dg2->addEdge ( putEdge( 1, 3, 0 ) );
dg2->addEdge ( putEdge( 2, 3, 0 ) );

// 3-layer graph ihas been created ...
printEdges( dg2->getEdges() );

Nodes* nodesBack;
nodesBack = dg2->topological_sort( 0 );
cout << "Nodes returned from topological sort: " << endl;
printNodes ( nodesBack );
```

Used in part 37b.

Finally, `prims` is tested on a graph of moderate size. This example is culled from [Cormen90], p.508.

⟨min_st test 39⟩ ≡

```
cout << "Working on mst..." << endl;
proxy_interface_var dg3 = fdg->createEmpty();

cout << "Placing nodes ... " << endl;
for( int i = 0; i < 9; ++i )
  dg3->addNode ( putNode( i ) );

cout << "Placing edges ... " << endl;
dg3->addEdge ( putEdge(  0, 1, 4 ) );
dg3->addEdge ( putEdge(  1, 2, 7 ) );
dg3->addEdge ( putEdge(  2, 3, 7 ) );
dg3->addEdge ( putEdge(  3, 4, 9 ) );
dg3->addEdge ( putEdge(  4, 5, 10 ) );
dg3->addEdge ( putEdge(  5, 6, 2 ) );
dg3->addEdge ( putEdge(  6, 7, 1 ) );
dg3->addEdge ( putEdge(  7, 0, 8 ) );
dg3->addEdge ( putEdge(  1, 7, 11 ) );
dg3->addEdge ( putEdge(  7, 8, 7 ) );
dg3->addEdge ( putEdge(  2, 8, 2 ) );
dg3->addEdge ( putEdge(  2, 5, 4 ) );
dg3->addEdge ( putEdge(  3, 5, 14 ) );
dg3->addEdge ( putEdge(  6, 8, 6 ) );

cout << "Calling prims ... " << endl;
proxy_interface_var dg4;// = fdg->createEmpty();
dg4 = dg3->prims( 0 );

cout << "Printing edges ... " << endl;
printEdges( dg4->getEdges() );
```

Used in part 37b.

The below calls demonstrate use of the edge and node iterators:

⟨client SII call iterator 40a⟩ ≡

```
cout << "Testing Iterator" << endl;
NodeIterator_var temp;
NodeType it;
Nodes* itList;
cout << "------------ getting one ---------" << endl;
temp = dg->getINodes();
if ( temp->next_one(it) )
  cout << it << endl;
else cout << "Returned nothing" << endl;

cout << "------------ getting five ---------" << endl;
temp->next_n(5, itList);
for (CORBA::ULong i = 0; i < itList->length(); ++i)
      cout << (*itList)[i] << endl;

cout << "------------ getting five ---------" << endl;
temp->next_n(5, itList);
for (CORBA::ULong i = 0; i < itList->length(); ++i)
      cout << (*itList)[i] << endl;

cout << "------------ getting all ---------" << endl;
temp->all(itList);
for (CORBA::ULong i = 0; i < itList->length(); ++i)
      cout << (*itList)[i] << endl;

cout << "------------ getting reset ---------" << endl;
temp->reset();
cout << "------------ getting five ---------" << endl;
temp->next_n(5, itList);
for (CORBA::ULong i = 0; i < itList->length(); ++i)
      cout << (*itList)[i] << endl;

cout << "End Iterator test" << endl;
```

Used in part 32a.

Next, the use of edge iterators is demonstrated. The results are compared those of an initial call to `getEdges`.

⟨client SII call edge stuff 40b⟩ ≡

⟨SII `getEdges` call 41a⟩
⟨SII Edge Iterator Tests misc. 41b⟩
⟨SII Edge Iterator reset 42a⟩

```
cout << "done with serial" << endl;
```

Used in part 32a.

⟨SII getEdges call 41a⟩ ≡

```
Edges* edgeList;
cout << "Trying getEdges..." << endl;
edgeList = dg->getEdges();
cout << "Call Get Edge complete" << endl;
cout << "Printing out return: " << edgeList->length() << endl;
printEdges( edgeList );
```

Used in part 40b.

⟨SII Edge Iterator Tests misc. 41b⟩ ≡

```
cout << "Testing Edge Iterator" << endl;
EdgeIterator_var e_temp;
EdgeType e_it;
Edges* e_itList;
cout << "------------ getting one ---------" << endl;
e_temp = dg->getIEdges();
if ( e_temp->next_one(e_it) )
   cout << "(" << e_it.node_i << "," << e_it.node_j <<
       "," << e_it.weight << ")" << endl;
else cout << "Returned nothing" << endl;

cout << "------------ getting five ---------" << endl;
e_temp->next_n(5, e_itList);
for (CORBA::ULong i = 0; i < e_itList->length(); ++i) {
      cout << "(" << (*e_itList)[i].node_i << "," <<
              (*e_itList)[i].node_j << "," <<
              (*e_itList)[i].weight << ")" << endl;
}
cout << "------------ getting five ---------" << endl;
e_temp->next_n(5, e_itList);
for (CORBA::ULong i = 0; i < e_itList->length(); ++i) {
      cout << "(" << (*e_itList)[i].node_i << "," <<
              (*e_itList)[i].node_j << "," <<
              (*e_itList)[i].weight << ")" << endl;
}
cout << "------------ getting all ---------" << endl;
e_temp->all(e_itList);
for (CORBA::ULong i = 0; i < e_itList->length(); ++i) {
      cout << "(" << (*e_itList)[i].node_i << "," <<
              (*e_itList)[i].node_j << "," <<
              (*e_itList)[i].weight << ")" << endl;
}
```

Used in part 40b.

⟨SII Edge Iterator reset 42a⟩ ≡

```
cout << "------------ getting reset ---------" << endl;
e_temp->reset();
cout << "------------ getting five ---------" << endl;
for (CORBA::ULong i = 0; i < e_itList->length(); ++i) {
      cout << "(" << (*e_itList)[i].node_i << "," <<
              (*e_itList)[i].node_j << "," <<
              (*e_itList)[i].weight << ")" << endl;
}
```

Used in part 40b.

A few other calls to the graph interface round out the SII tests:

⟨`client` SII call second series 42b⟩ ≡

```
cout << "Trying isConnected" << endl;
if ( dg->isConnected(1) )
      cout << "Connected" << endl;
if ( !( dg->isConnected(5) ) )
      cout << "Not Connected" << endl;

nodeList->length(3);
(*nodeList)[0] = 1;
(*nodeList)[1] = 2;
(*nodeList)[2] = 3;
cout << "Scalar returned by addWeights: "
      << dg->addWeights( (*nodeList) ) << endl;

stdGraphType * stgt = dg->getGraph();
cout << "Trying getGraph ---------------" << endl;
printEdges( &stgt->edgeList );
printNodes( &stgt->nodeList );
```

Used in part 32a.

Below is a concurrent call to a method returning a `sequence`. Notice that making this call requires more lines of code than do static calls, placing a greater burden on the implementor.

⟨client DII call node list return 43⟩ ≡

```
cout << "Trying to set up a concurrent call" << endl;
CORBA::Request_var _req = dg->_request("adjacentNodesOut");
_req->add_in_arg( "node_name" ) <<= (CORBA::ULong)node_num;
_req->set_return_type(_tc_Nodes);
_req->send_deferred();
cout << "Sent Call, hopefully enough postage" << endl;
_req->get_response();

if( _req->env()->exception() ) {
  CORBA::Exception *_ex = _req->env()->exception();
  CORBA::UnknownUserException *_uuex =
      CORBA::UnknownUserException::_downcast( _ex );
  if( _uuex )
    mico_throw( CORBA::UNKNOWN() );
  else mico_throw( *_ex );
}

Nodes* _res = new ::Nodes;
*_req->result()->value() >>= *_res;
nodeList = _res;

cout << "Deferred Call Adjacent Nodes Out complete" << endl;
cout << "Printing out return: " << nodeList->length() << endl;
for (CORBA::ULong i = 0; i < nodeList->length(); ++i)
      cout << (*nodeList)[i] << endl;
```

Used in part 32a.

Below is an example of a DII call to an interface method with a scalar return type. Notice that much of the code involved in the call is devoted to error-checking.

⟨client DII call scalar return 44⟩ ≡

```
cout << "Trying to set up a concurrent call" << endl;
nodeList->length(3);
(*nodeList)[0] = 1;
(*nodeList)[1] = 2;
(*nodeList)[2] = 3;
_req = dg->_request("addWeights");
_req->add_in_arg( "nodeList" ) <<= Nodes( (*nodeList) );
_req->set_return_type(_tc_Weight);
_req->send_deferred();

cout << "Sent Call, hopefully enough postage" << endl;
_req->get_response();

if( _req->env()->exception() ) {
  CORBA::Exception *_ex = _req->env()->exception();
  CORBA::UnknownUserException *_uuex =
      CORBA::UnknownUserException::_downcast( _ex );
  if( _uuex )
    mico_throw( CORBA::UNKNOWN() );
  else
    mico_throw( *_ex );
}

Weight _res_weight;
*_req->result()->value() >>= _res_weight;
cout << "Scalar returned by addWeights: "
      << _res_weight << endl;

 cout << "CLEANING UP";
 dg->destroy();
 dg2->destroy();
 dg3->destroy();
 dg4->destroy();
 e_temp->destroy(); //destroy the iterator
 temp->destroy(); //destroy the iterator
```

Used in part 32a.

5.3 Client Using DII

The below example client uses the CORBA Dynamic Invocation Interface to access server objects.

"dclient_ns.cc" 45a ≡
```
⟨client includes 32b⟩
⟨client helper functions 32c⟩
int main ( int argc, char * argv[]) {
  ⟨client orb setup 33d⟩
  ⟨client nameservices 34a⟩
  ⟨dclient setup code 45b⟩
  ⟨dclient algorithm invocation 46⟩
}
```

⟨dclient setup code 45b⟩ ≡
```
proxy_interface_var dg3 = fdg->createEmpty();

cout << "Placing nodes ... " << endl;
for( int i = 0; i < 9; ++i ) {
  cout << "Running with " << i << endl;
  dg3->addNode ( putNode( i ) );
}
cout << "Printing nodes ... " << endl;
printNodes( dg3->getNodes() );

cout << "Placing edges ... " << endl;
dg3->addEdge ( putEdge(  0, 1, 4 ) );
dg3->addEdge ( putEdge(  1, 2, 7 ) );
dg3->addEdge ( putEdge(  2, 3, 7 ) );
dg3->addEdge ( putEdge(  3, 4, 9 ) );
dg3->addEdge ( putEdge(  4, 5, 10 ) );
dg3->addEdge ( putEdge(  5, 6, 2 ) );
dg3->addEdge ( putEdge(  6, 7, 1 ) );
dg3->addEdge ( putEdge(  7, 0, 8 ) );
dg3->addEdge ( putEdge(  1, 7, 11 ) );

dg3->addEdge ( putEdge(  7, 8, 7 ) );
dg3->addEdge ( putEdge(  2, 8, 2 ) );
dg3->addEdge ( putEdge(  2, 5, 4 ) );
dg3->addEdge ( putEdge(  3, 5, 14 ) );
dg3->addEdge ( putEdge(  6, 8, 6 ) );

proxy_interface_var dg = fdg->clone( dg3 );
cout << "Allocated Graph" << endl;
```

Used in part 45a.

The below code demonstrates concurrent calls on the three main algorithms.

⟨dclient algorithm invocation 46⟩ ≡

```
cout << "Trying to set up multiple concurrent calls" << endl;
vector <CORBA::Request_var> _req(3);

⟨dclient request algorithms 47a⟩
⟨dclient get back responses 47b⟩
⟨dclient get back exceptions 47c⟩

CORBA::Boolean cycled;
*(_req[2])->result()->value() >>=
      CORBA::Any::to_boolean ( cycled ) ;

proxy_interface_var dg5;
*(_req[0])->result()->value() >>=  dg5;

Nodes * node_list = new Nodes;
*(_req[1])->result()->value() >>= *node_list;

if ( cycled ) cout << "It does have cycles";
else cout << "It does not have cycles";

cout << "Printing nodes ... " << endl;
printNodes( dg5->getNodes() );
cout << "Printing edges ... " << endl;
printEdges( dg5->getEdges() );

cout << " Printing Topological Sort results " << endl;
printNodes ( node_list );

cout << "CLEANING UP" << endl;
dg->destroy();
dg3->destroy();
dg5->destroy();
```

Used in part 45a.

⟨dclient request algorithms 47a⟩ ≡

```
        _req[0] = dg->_request("prims");
        _req[0]->add_in_arg( "r" ) <<= (CORBA::ULong)0;
        _req[0]->set_return_type(_tc_proxy_interface);
        _req[0]->send_deferred();
        cout << "Sent Call, hopefully enough postage" << endl;

        _req[1] = dg->_request("topological_sort");
        _req[1]->add_in_arg( "u" ) <<= (CORBA::ULong)0;
        _req[1]->set_return_type(_tc_Nodes);
        _req[1]->send_deferred();
        cout << "Sent Call, hopefully enough postage" << endl;

        _req[2] = dg3->_request("has_cycles");
        _req[2]->set_return_type(CORBA::_tc_boolean);
        _req[2]->send_deferred();
        cout << "Sent Call, hopefully enough postage" << endl;
```

Used in part 46.

Below, the client gets the results of the algorithm calls. A sample exception-handling routine is included.

⟨dclient get back responses 47b⟩ ≡

```
        _req[0]->get_response();
        _req[1]->get_response();
        _req[2]->get_response();
```

Used in part 46.

Below is an example of how to check the return code for exceptions that might have occurred during this call.

⟨dclient get back exceptions 47c⟩ ≡

```
        // Sturdy Exception Code just for _req[0]
        if( _req[0]->env()->exception() ) {
          CORBA::Exception *_ex = _req[0]->env()->exception();
          CORBA::UnknownUserException *_uuex =
              CORBA::UnknownUserException::_downcast( _ex );
          if( _uuex )
            mico_throw( CORBA::UNKNOWN() );
          else mico_throw( *_ex );
        }
```

Used in part 46.

5.4 Timing-Test Client

The below client implements a few timing tests comparing graph-service calls in a non-distributed as well as in a distributed setting.[4]

"tclient_ns.cc" 48a ≡
```
#include "matrixgraph.h"
#include "adjacencygraph.h"
⟨client includes 32b⟩
#include <mico/os-misc.h>
#include <iomanip.h>
⟨client helper functions 32c⟩
int main ( int argc, char * argv[]) {
  ⟨client orb setup 33d⟩
  ⟨client nameservices 34a⟩
  ⟨tclient setup test code 48b⟩
  ⟨tclient setup file 49⟩
  ⟨tclient addNodes individually 50⟩
  ⟨tclient addNodes all 51⟩
  ⟨tclient addEdges individually 52⟩
  ⟨tclient addEdges all 53⟩
  ⟨tclient cleanup 54⟩
}
```

⟨tclient setup test code 48b⟩ ≡
```
matrix_graph<long, long> * mg1;
adjacency_graph<long, long> * ag1;

matrix_graph<long, long> * mg2;
adjacency_graph<long, long> * ag2;


proxy_interface_var dg1;
proxy_interface_var dg2;
int size, increment;

fstream outFile;

cout << "Enter the maximum size of the test: ";
cin >> size;
cout << "Enter the increment for the test: ";
cin >> increment;

outFile.open("times.txt", ios::app);
```

Used in part 48a.

[4]The OSMisc::gettime calls are taken from the examples/bench directory of the MICO 2.2.5 release.

Below, the time taken for a series of calls to `addNode` and `addEdge` is calculated in both a distributed and non-distributed setting.

⟨`tclient` setup file 49⟩ ≡

```
        outFile << "GT\t"
                << "Size\t"
                << "LSN\t" << "DSN\t" << "LMN\t" << "DMN\t"
                << "LSE\t" << "DSE\t" << "LME\t" << "DME"
                << endl;

        CORBA::Long t1, t2;
        CORBA::Double lsn, dsn, lmn, dmn;
        CORBA::Double lse, dse, lme, dme;
        vector<int> stlnlist(size);
        vector<int>::iterator ni;
        Nodes corbanlist;
        corbanlist.length(size);
        vector<pair<int, int> > stlelist(size);
        vector<pair<int, int> >::iterator ei;
        Edges corbaelist;
        corbaelist.length(size);
```

Used in part 48a.

⟨`tclient` addNodes individually 50⟩ ≡

```
for (int idx = 1; idx <= size; idx+=increment) {
  mg1 = new matrix_graph<long,long>;
  ag1 = new adjacency_graph<long,long>;
  mg2 = new matrix_graph<long,long>;
  ag2 = new adjacency_graph<long,long>;
  dg1 = fdg->createEmpty();
  dg2 = fdg->createEmpty();
  corbanlist.length(idx);
  corbaelist.length(idx);

  if ( input == 0 ) {
     outFile << "adj\t";
  } else {
     outFile << "mat\t";
  }

  outFile << idx << "\t";
  outFile << setiosflags(ios::fixed)
          << setprecision(3);

  if (input > 0) {
    t1 = OSMisc::gettime();
    for (int i = 0; i < idx; ++i)
      mg1->addNode(i);
    t2 = OSMisc::gettime();
  } else {
    t1 = OSMisc::gettime();
    for (int i = 0; i < idx; ++i)
      ag1->addNode(i);
    t2 = OSMisc::gettime();
  }
  lsn = (double)(t2-t1)/idx;

  t1 = OSMisc::gettime();
  for (int i = 0; i < idx; ++i)
    dg1->addNode(i);
  t2 = OSMisc::gettime();
  dsn = (double)(t2-t1)/idx;
```

Used in part 48a.

⟨**tclient** addNodes all 51⟩ ≡

```
ni = stlnlist.begin();
if (input > 0) {
   t1 = OSMisc::gettime();
   for (int i = 0; i < idx; ++i, ++ni)
     stlnlist[i] = i;
   mg2->addNodes(stlnlist.begin(), ni);
   t2 = OSMisc::gettime();
 } else {
   t1 = OSMisc::gettime();
   for (int i = 0; i < idx; ++i, ++ni)
     stlnlist[i] = i;
   ag2->addNodes(stlnlist.begin(), ni);
   t2 = OSMisc::gettime();
 }
 lmn = (double)(t2-t1)/idx;

 t1 = OSMisc::gettime();
 for (int i = 0; i < idx; ++i)
   corbanlist[i] = i;
 dg2->addNodes( corbanlist );
 t2 = OSMisc::gettime();
 dmn = (double)(t2-t1)/idx;
```

Used in part 48a.

⟨**tclient** addEdges individually 52⟩ ≡

```
if (input > 0) {
  t1 = OSMisc::gettime();
  for (int i = 0; i < idx; ++i) {
     mg1->addEdge( i, i+1, 3000);
     mg1->addEdge( i+1, i, 3000);
  }
  t2 = OSMisc::gettime();
} else {
  t1 = OSMisc::gettime();
  for (int i = 0; i < idx; ++i) {
     ag1->addEdge( i, i+1, 3000);
     ag1->addEdge( i+1, i, 3000);
  }
  t2 = OSMisc::gettime();
}
lse = (double)(t2-t1)/idx;

t1 = OSMisc::gettime();
for (int i = 0; i < idx; ++i) {
  dg1->addEdge( putEdge (i, i+1, 3000));
  dg1->addEdge( putEdge (i+1, i, 3000));
}
t2 = OSMisc::gettime();
dse = (double)(t2-t1)/idx;
```

Used in part 48a.

⟨`tclient` addEdges all 53⟩ ≡

```
ei = stlelist.begin();
if (input > 0) {
  t1 = OSMisc::gettime();
  for (int i = 0; i < idx; ++i, ++ei ) {
    stlelist[i].first = i;
    stlelist[i].second = i + 1;
  }
  mg2->addEdges(stlelist.begin(), ei);
  t2 = OSMisc::gettime();
} else {
  t1 = OSMisc::gettime();
  for (int i = 0; i < idx; ++i, ++ei ) {
    stlelist[i].first = i;
    stlelist[i].second = i + 1;
  }
  ag2->addEdges(stlelist.begin(), ei);
  t2 = OSMisc::gettime();
}
lme = (double)(t2-t1)/idx;

t1 = OSMisc::gettime();
for (int i = 0; i < idx; ++i)
  corbaelist[i] = putEdge ( i, i+1, 3000 );
dg2->addEdges( corbaelist );
t2 = OSMisc::gettime();

dme = (double)(t2-t1)/idx;
```

Used in part 48a.

This is where the transcription to the log is performed as well as the release of memory allocated during this run:

⟨tclient cleanup 54⟩ ≡

```
    outFile << lsn << "\t" <<  dsn << "\t";
    outFile << lmn << "\t" <<  dmn << "\t";
    outFile << lse << "\t" <<  dse << "\t";
    outFile << lme << "\t" <<  dme << "\t" << endl;
    dg1->destroy();
    dg2->destroy();
    delete mg1;
    delete ag1;
    delete ag2;
    delete mg2;
  }

  outFile.close();
```

Used in part 48a.

5.4.1 Timing Results

A comparison of times for calls on distributed graphs versus those for their non-distributed counterparts has been obtained by running the `tclient`. We list the results of these tests by the appropriate remote computer name. The computers involved in these tests were `resnet-991`, `cory`, and `dishwasher`.

- `cory` is a Intel Pentium II machine clocked at 333Mhz running Linux 2.2.4 with 64M of memory and 128 swap.
- `dishwasher` is a UltraSparc machine with dual 200Mhz CPUs running Sun Solaris 5.6 with 400M of memory and 700 swap.
- `resnet-991` is a Cyrix 586 machine clocked at 166Mhz running Linux 2.2.4 with 128M and 128 swap.

5.4.2 Testbed

For an evaluation of performance across a wide range of graph sizes, we have run these tests incrementally on graphs of varying sizes. Graphs with numbers of nodes incremented by 100 (up to a maximum of 2000) were tested. Running these tests on such a wide spectrum of sizes helps to smooth out data anomalies that may be caused by caching or other architectural "sweet spots"[Patterson96].

For each increase in size over the period of calculation, eight distinct tests were performed.

- Local Single Add Node (`lsn`) invokes the `addNode` method to add nodes one at a time to a local graph.
- Distributed Single Add Node (`dsn`) invokes the `addNode` method to add nodes one at a time to a distributed graph.

- Local Multiple Add Node (`lmn`) creates a `vector` of nodes and then calls the `addNodes` method for a local graph.
- Distributed Multiple Add Node (`dmn`) creates a `sequence` of nodes and calls the `addNodes` method for a distributed graph.
- Local Single Add Edge (`lse`) invokes the `addEdge` method to add edges one at a time to a local graph.
- Distributed Single Add Edge (`dse`) invokes the `addEdge` method to add edges one at a time to a distributed graph.
- Local Multiple Add Edge (`lme`) creates a `vector` of edges and then calls the `addEdges` method for a local graph.
- Distributed Multiple Add Edge (`dme`) creates a `sequence` of edges and calls the `addEdges` method for a distributed graph.

Each group of tests was run with the three different machines acting as servers. The client always originated from `resnet-991`. The structure of the files presented below is a `ping` listing from `resnet-991` to the host, `traceroute` from `resnet-991` to the host, and then the complete test for that host generated by two runs of the client on `resnet-991`. The client is run to generate a listing for the matrix and the adjacency version.

5.4.3 cory raw results

⟨cory-net 55⟩ ≡

```
PING cory.ml.org (24.29.44.241): 56 data bytes
64 bytes from 24.29.44.241: icmp_seq=0 ttl=242 time=42.537 ms
64 bytes from 24.29.44.241: icmp_seq=1 ttl=242 time=48.581 ms
64 bytes from 24.29.44.241: icmp_seq=2 ttl=242 time=49.688 ms
64 bytes from 24.29.44.241: icmp_seq=3 ttl=242 time=40.990 ms
64 bytes from 24.29.44.241: icmp_seq=4 ttl=242 time=42.132 ms
--- cory.ml.org ping statistics ---
5 packets transmitted, 5 packets received, 0% packet loss
round-trip min/avg/max = 40.990/44.785/49.688 ms
 1  128.113.180.254 (128.113.180.254)  1.225 ms  0.942 ms  0.812 ms
 2  at-gw2-alb-2-0-t3.appliedtheory.net (169.130.22.5)  3.710 ms  3.613 ms  4.177 ms
 3  at-gw2-syr-1-1-0-t3.appliedtheory.net (169.130.1.22)  5.284 ms  6.425 ms  5.548 ms
 4  at-bb1-pen-5-0-0-t3.appliedtheory.net (169.130.1.121)  17.954 ms  18.490 ms  22.472 ms
 5  sl-bb10-pen-4-2.sprintlink.net (144.232.5.109)  25.214 ms  19.062 ms  19.715 ms
 6  sl-bb6-pen-0-0-0.sprintlink.net (144.232.5.46)  23.939 ms  42.087 ms  21.176 ms
 7  core4-hssi5-0-0.westorange.cw.net (206.157.77.105)  21.874 ms  20.555 ms  19.374 ms
 8  core1.westorange.cw.net (204.70.4.181)  32.296 ms  23.044 ms  24.492 ms
 9  border3-fddi-0.westorange.cw.net (204.70.64.19)  25.774 ms  30.678 ms  26.669 ms
10  road-runner-group.westorange.cw.net (204.70.67.6)  31.514 ms  28.192 ms  28.051 ms
11  router-105.nycap.rr.com (24.29.32.105)  33.637 ms  31.948 ms  29.101 ms
12  router-122.nycap.rr.com (24.29.32.122)  36.331 ms  38.181 ms  43.885 ms
13  cory.ml.org (24.29.44.241)  42.988 ms  46.640 ms  40.154 ms
```

Not used.

⟨resnet-cory 56⟩ ≡

```
GT    Size  LSN    DSN    LMN    DMN    LSE    DSE     LME    DME
mat   1     0.000  71.000 0.000  74.000 0.000  138.000 0.000  60.000
mat   101   0.743  57.317 0.733  0.881  1.079  114.089 0.653  1.634
mat   201   1.488  52.169 1.478  0.701  2.114  110.692 1.264  1.085
mat   301   2.296  63.326 2.229  0.920  3.272  106.066 1.990  1.156
mat   401   3.409  56.282 2.988  1.052  4.646  115.097 2.521  1.698
mat   501   3.844  62.852 3.818  1.251  5.639  129.521 3.154  2.092
mat   601   4.649  59.579 4.834  1.443  6.885  118.092 4.088  1.712
mat   701   5.632  63.586 5.358  1.803  7.870  115.489 6.381  2.367
mat   801   8.383  67.082 7.511  2.939  11.245 122.060 10.295 2.426
mat   901   16.320 53.576 7.708  2.226  10.242 107.311 6.089  2.486
mat   1001  8.366  53.621 8.276  2.324  11.461 105.373 6.440  2.776
mat   1101  8.857  51.721 8.604  2.716  12.665 107.154 7.681  7.182
mat   1201  9.604  52.868 9.703  2.870  14.026 109.172 8.436  3.853
mat   1301  10.592 50.822 10.193 3.091  15.064 105.163 8.842  3.940
mat   1401  11.318 53.859 11.110 3.359  16.166 101.046 9.502  4.035
mat   1501  12.274 50.138 12.132 3.528  16.763 101.618 9.987  4.026
mat   1601  13.128 53.416 12.811 3.761  18.037 95.415  10.650 4.011
mat   1701  13.790 48.258 13.518 4.093  18.692 96.671  11.075 5.081
mat   1801  14.552 47.966 14.470 4.306  19.536 96.786  11.781 5.239
mat   1901  15.463 50.208 15.316 4.435  20.742 98.736  12.292 6.170
GT    Size  LSN    DSN    LMN    DMN    LSE    DSE     LME    DME
adj   1     0.000  64.000 1.000  47.000 0.000  87.000  0.000  48.000
adj   101   0.495  47.683 0.485  0.713  0.891  113.495 2.099  1.624
adj   201   0.955  51.900 0.935  0.542  1.677  96.269  0.841  0.856
adj   301   1.395  51.472 1.415  0.635  2.522  97.963  1.266  1.551
adj   401   1.865  48.406 1.843  0.713  3.332  98.202  1.663  1.000
adj   501   2.335  50.124 2.301  1.096  4.166  109.880 2.118  1.459
adj   601   2.792  49.987 2.797  0.980  5.012  110.280 4.910  1.835
adj   701   6.762  50.807 6.605  1.146  12.121 97.659  5.786  1.655
adj   801   4.134  48.920 3.991  1.275  7.181  107.899 3.443  1.836
adj   901   4.458  55.075 5.754  1.436  10.788 105.916 6.538  2.061
adj   1001  7.551  48.537 7.559  1.616  13.253 123.426 6.840  2.329
adj   1101  7.379  50.650 7.370  1.824  13.183 114.185 6.937  2.166
adj   1201  8.469  56.296 6.197  1.859  11.457 106.255 5.689  2.388
adj   1301  6.905  52.920 6.825  2.017  12.701 128.406 5.978  2.673
adj   1401  7.142  61.130 6.983  2.177  12.472 113.182 6.201  2.498
adj   1501  7.270  58.957 7.145  2.363  13.417 105.845 6.654  2.709
adj   1601  7.799  55.438 7.770  2.455  14.252 107.012 7.116  2.851
adj   1701  8.320  57.505 8.467  2.631  15.688 121.429 7.761  3.212
adj   1801  8.988  60.951 8.644  2.867  16.544 133.363 8.309  3.421
adj   1901  9.206  74.414 9.258  2.995  17.385 134.521 8.457  3.319
```

Not used.

5.4.4 dishwasher raw results

⟨dishwasher-net 57⟩ ≡

```
PING dishwasher.cs.rpi.edu (128.213.8.31): 56 data bytes
64 bytes from 128.213.8.31: icmp_seq=0 ttl=253 time=1.937 ms
64 bytes from 128.213.8.31: icmp_seq=1 ttl=253 time=1.454 ms
64 bytes from 128.213.8.31: icmp_seq=2 ttl=253 time=3.466 ms
64 bytes from 128.213.8.31: icmp_seq=3 ttl=253 time=1.420 ms
64 bytes from 128.213.8.31: icmp_seq=4 ttl=253 time=20.279 ms
--- dishwasher.cs.rpi.edu ping statistics ---
5 packets transmitted, 5 packets received, 0% packet loss
round-trip min/avg/max = 1.420/5.711/20.279 ms
 1  128.113.180.254 (128.113.180.254)  1.434 ms  0.849 ms  0.819 ms
 2  cs-gw.rpi.edu (128.113.113.252)  1.063 ms  1.353 ms  1.335 ms
 3  dishwasher.cs.rpi.edu (128.213.8.31)  1.517 ms *  1.478 ms
```

Not used.

⟨resnet-dishwasher 58⟩ ≡

GT	Size	LSN	DSN	LMN	DMN	LSE	DSE	LME	DME
mat	1	0.000	107.000	0.000	83.000	0.000	62.000	0.000	151.000
mat	101	0.743	4.950	0.743	0.772	1.079	10.168	0.703	0.881
mat	201	1.582	5.701	1.557	1.189	3.269	10.657	1.338	1.249
mat	301	2.468	7.598	2.349	1.721	3.412	13.183	2.269	1.674
mat	401	3.057	8.085	2.990	2.501	4.693	15.000	2.519	2.067
mat	501	4.172	7.136	4.058	2.840	5.523	13.164	3.082	2.363
mat	601	4.602	8.218	4.805	3.582	6.784	14.940	3.992	3.017
mat	701	5.608	8.535	5.536	3.979	7.668	14.738	4.546	3.258
mat	801	6.362	9.433	6.383	4.508	8.924	16.504	5.461	3.729
mat	901	7.301	9.592	7.273	5.173	9.857	16.406	5.830	3.994
mat	1001	7.903	10.828	7.784	5.742	10.843	17.473	6.387	4.347
mat	1101	8.845	10.825	8.690	6.220	12.758	18.768	7.766	5.685
mat	1201	9.606	13.307	9.623	6.798	13.894	20.022	8.302	5.384
mat	1301	10.666	14.248	10.666	7.613	15.208	20.699	9.059	5.769
mat	1401	11.373	14.805	11.248	7.970	16.551	21.460	9.758	6.171
mat	1501	12.428	13.943	12.313	8.709	17.149	21.911	10.334	6.554
mat	1601	13.246	14.581	12.901	9.273	18.166	23.407	10.588	7.014
mat	1701	13.774	15.713	14.192	9.918	19.404	22.526	11.440	7.225
mat	1801	14.706	17.364	14.514	10.758	20.233	23.581	11.827	8.128
mat	1901	15.601	19.058	15.516	10.951	22.082	25.476	12.460	7.874
GT	Size	LSN	DSN	LMN	DMN	LSE	DSE	LME	DME
adj	1	0.000	123.000	0.000	36.000	0.000	72.000	0.000	20.000
adj	101	0.505	4.545	0.485	0.515	0.891	9.634	0.446	0.723
adj	201	0.950	4.841	0.945	0.711	1.786	11.363	0.851	0.960
adj	301	1.395	5.306	1.405	0.997	2.495	12.415	1.286	1.262
adj	401	1.925	6.409	1.860	1.461	3.342	13.514	1.678	1.516
adj	501	2.337	6.321	2.299	1.529	4.176	12.948	2.080	1.822
adj	601	2.814	6.569	2.797	1.827	5.037	15.815	2.539	2.136
adj	701	3.305	9.098	3.465	2.185	6.247	15.939	3.170	2.465
adj	801	3.940	7.212	3.698	2.426	6.955	15.934	3.678	2.664
adj	901	4.434	7.370	4.401	2.706	7.992	16.829	3.980	3.009
adj	1001	4.816	7.628	4.773	3.030	8.852	15.896	4.397	3.246
adj	1101	5.279	7.894	5.254	3.280	9.720	16.754	4.809	3.526
adj	1201	5.846	8.490	5.689	5.098	10.721	18.607	5.254	3.938
adj	1301	6.312	10.661	6.158	3.860	11.542	21.723	5.658	4.151
adj	1401	6.773	10.380	6.599	4.403	12.321	22.786	6.203	4.406
adj	1501	7.672	12.231	7.544	4.532	13.629	24.271	6.726	4.913
adj	1601	7.690	11.200	7.724	4.780	14.162	21.771	7.102	5.051
adj	1701	8.210	9.976	8.094	5.115	27.857	22.279	15.069	5.685
adj	1801	17.310	11.623	13.096	5.510	16.064	22.074	8.077	5.931
adj	1901	9.328	10.855	9.095	5.656	17.052	22.942	8.485	6.024

Not used.

5.4.5 resnet-991 raw results

⟨local-net 59⟩ ≡

```
PING resnet-991.dynamic.rpi.edu (128.113.180.36): 56 data bytes
64 bytes from 128.113.180.36: icmp_seq=0 ttl=255 time=0.419 ms
64 bytes from 128.113.180.36: icmp_seq=1 ttl=255 time=0.263 ms
64 bytes from 128.113.180.36: icmp_seq=2 ttl=255 time=0.268 ms
64 bytes from 128.113.180.36: icmp_seq=3 ttl=255 time=0.289 ms
64 bytes from 128.113.180.36: icmp_seq=4 ttl=255 time=0.273 ms
--- resnet-991.dynamic.rpi.edu ping statistics ---
5 packets transmitted, 5 packets received, 0% packet loss
round-trip min/avg/max = 0.263/0.302/0.419 ms
 1  resnet-991 (128.113.180.36)  0.379 ms  0.239 ms  0.214 ms
```

Not used.

⟨resnet-local 60⟩ ≡

GT	Size	LSN	DSN	LMN	DMN	LSE	DSE	LME	DME
mat	1	0.000	8.000	0.000	7.000	1.000	11.000	0.000	7.000
mat	101	0.752	4.950	0.733	0.851	1.089	10.535	0.653	1.218
mat	201	1.493	6.886	1.602	1.622	2.169	11.796	1.259	1.816
mat	301	2.870	7.150	2.239	2.372	3.399	13.399	2.003	3.093
mat	401	3.429	7.167	3.486	3.030	4.197	13.958	2.938	3.050
mat	501	3.796	8.226	3.792	4.086	5.160	15.226	3.078	3.597
mat	601	5.245	9.178	4.987	4.854	6.799	16.809	3.963	4.762
mat	701	5.601	9.705	5.524	5.509	7.769	18.639	5.010	5.492
mat	801	6.684	10.483	6.300	6.265	8.960	18.453	5.079	5.919
mat	901	7.303	11.394	7.032	6.994	10.098	19.432	5.848	6.378
mat	1001	8.075	12.161	8.138	7.859	10.933	20.201	6.566	6.997
mat	1101	9.001	13.118	8.763	8.695	12.513	22.342	7.759	8.337
mat	1201	9.659	13.784	9.505	9.492	14.831	23.884	8.498	8.995
mat	1301	10.528	15.042	10.690	10.355	14.513	24.915	9.214	9.601
mat	1401	11.510	15.897	11.278	11.391	15.997	25.835	10.426	10.502
mat	1501	12.389	16.526	12.039	12.063	17.178	26.773	10.269	10.793
mat	1601	13.507	17.467	12.873	13.037	18.129	28.167	10.628	12.919
mat	1701	13.892	17.797	13.708	13.518	18.695	29.513	11.372	13.918
mat	1801	14.672	19.095	14.741	14.504	20.122	30.078	12.110	13.319
mat	1901	15.727	20.038	15.482	15.294	21.197	30.957	12.396	16.346
GT	Size	LSN	DSN	LMN	DMN	LSE	DSE	LME	DME
adj	1	0.000	8.000	0.000	8.000	0.000	12.000	0.000	7.000
adj	101	0.505	4.792	0.495	0.604	2.485	11.129	0.436	1.020
adj	201	1.005	5.348	0.945	1.050	1.697	12.990	0.841	1.393
adj	301	1.399	5.748	1.405	1.492	2.993	12.179	1.252	1.797
adj	401	2.227	6.147	1.848	1.950	3.713	13.436	1.673	2.219
adj	501	2.357	6.962	2.305	2.425	4.509	14.343	2.074	3.002
adj	601	2.805	7.421	2.767	3.138	5.012	15.383	2.509	3.386
adj	701	3.262	7.856	3.441	3.391	6.124	16.371	3.240	3.656
adj	801	3.929	8.287	3.885	3.869	6.943	17.276	3.562	4.019
adj	901	4.371	8.909	4.163	4.485	7.851	18.249	3.790	4.612
adj	1001	4.878	9.381	4.754	5.005	8.816	18.931	4.378	5.081
adj	1101	5.306	9.855	5.243	5.600	10.399	20.606	4.949	5.736
adj	1201	5.888	10.515	5.937	6.008	10.908	21.423	5.389	5.994
adj	1301	6.512	11.158	6.144	6.470	11.580	22.064	5.636	6.455
adj	1401	6.670	11.477	6.583	6.970	12.490	22.910	6.178	6.808
adj	1501	7.201	11.873	7.169	7.400	13.264	23.883	6.598	7.349
adj	1601	7.856	12.663	7.856	8.067	14.510	25.372	7.272	7.957
adj	1701	8.489	13.095	8.638	8.551	15.560	26.246	7.755	8.451
adj	1801	8.913	13.831	8.823	9.110	16.441	26.857	8.000	8.761
adj	1901	9.198	13.963	9.063	9.366	17.463	28.373	8.692	9.375

Not used.

5.4.6 Test Results

The results of our timing tests reveal that adding nodes and edges individually is much more expensive than adding sequences of nodes and edges. Furthermore, they show that distributing computation to a faster remote computer yields significant performance benefits.[5]

When the tests are run entirely on a single local system, the performance of that system degrades in a linear fashion as the CPU is tasked to maintain both a client and a server. When the server is moved off of the local system to `cory`, advantages are gained due to the faster processing speed of the remote machine—these advantages become more striking with larger numbers of nodes and edges.

When the server is moved to `dishwasher`, the relatively small network latency, along with the high speed of that machine's processor, yields high performance in the `DMN` and `DME` tests.

6 A Simple Interactive Client

The below interactive client demonstrates the sort of data-mapping that a client will likely provide.

6.1 Overview of the Source File

"iclient_ns.cc" 61 ≡

```
⟨ic includes etc 62⟩
⟨client helper functions 32c⟩

class nodeMapper {
  ⟨ic nodeMapper innards 63⟩
};

int main ( int argc, char * argv[]) {
  ⟨ic data setup 64a⟩
  ⟨client orb setup 33d⟩
  ⟨client nameservices 34a⟩

  dg = fdg->create( (*graph) );
  cout << "Welcome. System is on line." << endl << endl;
  while( true ) {
    ⟨ic handle command line entries 64b⟩
  }
  cout << endl << "Have a nice day!" << endl << endl;
  return 0;
} // end main()

⟨ic grab_tokens function 69b⟩
⟨ic help function— 70a⟩
```

[5]Incidentally, although no formal performance metrics were performed, the distributed graph was able to allocate at least 16000 nodes and edges in a sparse graph in which each node is connected to the first node.

6.2 Implementation

⟨ic includes etc 62⟩ ≡

```
#include<iostream>
#include<string>
#include<vector>
#include<map>

⟨client includes 32b⟩

// forward refs to helper functions
vector<string> grab_tokens( bool& valid, int x );
void help();
```

Used in part 61.

The below class provides for mapping of the integer data manipulated by our distributed graph system to data of another type (in this case, that type is `string`).

6.3 Mapping Nodes to Data

⟨ic nodeMapper innards 63⟩ ≡

```
private:
  typedef map<int, string> TableType;
  TableType table;
  int nextIndex;
public:
  nodeMapper() : nextIndex(0), table() {}
  ~nodeMapper() {}

  void insert_node( string x ) {
    table[nextIndex] = x;
    ++nextIndex;
  }
  void remove_node( int x ) {
    TableType::iterator i = table.find( x );
    if( i != table.end() )
        table.erase( i );
  }
  int size() const {
    return table.size();
  }
  string int_to_string( const int& x ) {
    TableType::iterator i = table.find( x );
    if( i != table.end() )
      return (*i).second;
    else return string("");
  }
  int string_to_int( const string& x ) const {
    TableType::const_iterator i;
    for( i = table.begin(); i != table.end(); ++i )
      if( (*i).second == x )
        return (*i).first;
  }
  bool name_exists( const string& x ) const {
    TableType::const_iterator i;
    for( i = table.begin(); i != table.end(); ++i )
      if( (*i).second == x )
        return true;
    return false;
  }
  bool int_exists( const int& x ) const {
    return( table.find( x ) != table.end() );
  }
```

Used in part 61.

6.4 Main Program

⟨ic data setup 64a⟩ ≡

```
nodeMapper lookup;

vector<string> lineIn;
string tokenIn;
bool valid;  // was it a valid command line?
bool badcmd;
vector<string> moreTokens;

proxy_interface_var dg;
stdGraphType* graph = new stdGraphType;

Nodes* nodeList;
Edges* edgeList;
```

Used in part 61.

Requested services are performed on a single graph. All command lines must be terminated with the character '`!`'.

⟨ic handle command line entries 64b⟩ ≡
⟨ic reset prompt 64c⟩
⟨ic addedge request 65a⟩
⟨ic rmedge request 65b⟩
⟨ic addnode request 65c⟩
⟨ic rmnode request 66a⟩
⟨ic isconn request 66b⟩
⟨ic adjo request 66c⟩
⟨ic adji request 67a⟩
⟨ic ptnodes request 67b⟩
⟨ic ptedges request 67c⟩
⟨ic cycle request 68a⟩
⟨ic topsort request 68b⟩
⟨ic help request 68c⟩
⟨ic quit request 68d⟩
⟨ic handle bad command line 69a⟩

Used in part 61.

⟨ic reset prompt 64c⟩ ≡

```
moreTokens.erase( moreTokens.begin(), moreTokens.end() );
badcmd = false;
cout << "bkp> ";
cin >> tokenIn;
```

Used in part 64b.

6.4.1 Making Proxy Calls

It is in the below code that the calls to the distributed graph proxy are made. Some checking of entered data is performed locally before these calls are made.

⟨ic addedge request 65a⟩ ≡

```
if( tokenIn == "addedge" ) {
  moreTokens = grab_tokens(valid, 2);
  if( lookup.name_exists( moreTokens[0] )
      && lookup.name_exists( moreTokens[1] ) ) {
    dg->addEdge( putEdge(
          lookup.string_to_int( moreTokens[0] ),
          lookup.string_to_int( moreTokens[1] ),
          (CORBA::ULong)(0) ));
  }
  else cout << "Cannot add edge to/from"
            << " nonexistent node!" << endl;
}
```

Used in part 64b.

⟨ic rmedge request 65b⟩ ≡

```
else if( tokenIn == "rmedge" ) {
  moreTokens = grab_tokens(valid, 2);
  if( lookup.name_exists( moreTokens[0] )
      && lookup.name_exists( moreTokens[1] ) ) {
    dg->removeEdge( lookup.string_to_int( moreTokens[0] ),
                    lookup.string_to_int( moreTokens[1] ) );
  }
  else cout << "That edge does not exist!" << endl;
}
```

Used in part 64b.

⟨ic addnode request 65c⟩ ≡

```
else if( tokenIn == "addnode" ) {
  moreTokens = grab_tokens(valid, 1);
  if( !lookup.name_exists( moreTokens[0] ) ) {
    lookup.insert_node( moreTokens[0] );
    dg->addNode( lookup.string_to_int( moreTokens[0] ) );
  }
  else cout << "That node already exists!" << endl;
}
```

Used in part 64b.

⟨ic rmnode request 66a⟩ ≡

```
else if( tokenIn == "rmnode" ) {
  moreTokens = grab_tokens(valid, 1);
  if( lookup.name_exists( moreTokens[0] ) ) {
    Nodes tmpEdges(1);
    tmpEdges[0] = lookup.string_to_int( moreTokens[0] );
    lookup.remove_node(
      lookup.string_to_int( moreTokens[0] ) );
    dg->removeNodes( tmpEdges );
  }
  else cout << "Node doesn't exist!!" << endl;
}
```

Used in part 64b.

⟨ic isconn request 66b⟩ ≡

```
else if( tokenIn == "isconn" ) {
  moreTokens = grab_tokens(valid, 1);
  if( lookup.name_exists( moreTokens[0] ) ) {
     if( dg->isConnected(
           lookup.string_to_int( moreTokens[0] ) ) )
       cout << "Yes, that node is connected.";
     else
       cout << "No, that node is not connected.";
     cout << endl;
  }
  else cout << "Node doesn't exist!!" << endl;
}
```

Used in part 64b.

⟨ic adjo request 66c⟩ ≡

```
else if( tokenIn == "adjo" ) {
  moreTokens = grab_tokens(valid, 1);
  if( lookup.name_exists( moreTokens[0] ) ) {
    nodeList = dg->adjacentNodesOut( lookup.string_to_int(
                           moreTokens[0] ) );
    for( int i = 0; i < (*nodeList).length(); ++i )
      cout << lookup.int_to_string( (*nodeList)[i] ) << endl;
  }
  else cout << "Node doesn't exist!!" << endl;
}
```

Used in part 64b.

⟨ic adji request 67a⟩ ≡

```
else if( tokenIn == "adji" ) {
  moreTokens = grab_tokens(valid, 1);
  if( lookup.name_exists( moreTokens[0] ) ) {
    nodeList = dg->adjacentNodesIn( lookup.string_to_int(
                                    moreTokens[0] ) );
    for( int i = 0; i < (*nodeList).length(); ++i )
      cout << lookup.int_to_string( (*nodeList)[i] ) << endl;
  }
  else cout << "Node doesn't exist!!" << endl;
}
```

Used in part 64b.

⟨ic ptnodes request 67b⟩ ≡

```
else if( tokenIn == "ptnodes" ) {
  grab_tokens(valid, 0);
  nodeList = dg->getNodes();
  for( int i = 0; i < (*nodeList).length(); ++i ) {
    string node1( lookup.int_to_string( (*nodeList)[i] ));
    if( lookup.name_exists( node1 ) )
      cout << lookup.int_to_string( (*nodeList)[i] ) << endl;
  }
  delete nodeList;
}
```

Used in part 64b.

⟨ic ptedges request 67c⟩ ≡

```
else if( tokenIn == "ptedges" ) {
  grab_tokens(valid, 0);
  edgeList = dg->getEdges();
  for( int i = 0; i < (*edgeList).length(); ++i ) {
    string node1 = lookup.int_to_string(
                           (*edgeList)[i].node_i );
    string node2 = lookup.int_to_string(
                           (*edgeList)[i].node_j );
    if( lookup.name_exists( node1 )
      && lookup.name_exists( node2 ) )
        cout << node1 << " --> " << node2 << endl;
  }
  delete edgeList;
}
```

Used in part 64b.

⟨ic cycle request 68a⟩ ≡

```
else if( tokenIn == "cycle?" ) {
  grab_tokens(valid, 0);
  if( dg->has_cycles() )
    cout << "The graph has a cycle.";
  else cout << "The graph has no cycles.";
  cout << endl;
}
```

Used in part 64b.

⟨ic topsort request 68b⟩ ≡

```
else if( tokenIn == "topsort?" ) {
  moreTokens = grab_tokens(valid, 1);
  if( lookup.name_exists( moreTokens[0] ) ) {
    nodeList = dg->topological_sort(
          lookup.string_to_int( moreTokens[0] ) );
    for( int i = 0; i < (*nodeList).length(); ++i )
      cout << lookup.int_to_string( (*nodeList)[i] ) << endl;
    delete nodeList;
  }
  else cout << "Node doesn't exist!!" << endl;
}
```

Used in part 64b.

⟨ic help request 68c⟩ ≡

```
else if( tokenIn == "help" ) {
  grab_tokens(valid, 0);
  help();
  valid = true; // allow other toks
}
```

Used in part 64b.

⟨ic quit request 68d⟩ ≡

```
else if( tokenIn == "quit" ) {
  grab_tokens(valid, 0); // all done
  break;
}
```

Used in part 64b.

⟨ic handle bad command line 69a⟩ ≡

```
else {
  grab_tokens(valid, 0);
  badcmd = true;
  cout << endl << "!!error!! Command "
       << tokenIn << " not recognized!" << endl << endl;
}
if( !valid && !badcmd )
  cout << endl << "!!error!! Bad input line!" << endl << endl;
```

Used in part 64b.

Finally, the below utility methods are part of this client program:

6.5 Utilities

⟨ic `grab_tokens` function 69b⟩ ≡

```
vector<string> grab_tokens( bool& valid, const int x ) {
  vector<string> tmpTokens;
  string curToken("");
  while( curToken != "!" ) {
    cin >> curToken;
    if( curToken != "!" )
      tmpTokens.push_back( curToken );
  }
  if( tmpTokens.size() == x )
    valid = true;
  else valid = false;

  return tmpTokens;
}
```

Used in part 61.

⟨ic help function— 70a⟩ ≡

```
void help() {
  cout << endl << endl;

  cout << "Valid commands are: " << endl;
  cout << "    addedge # #" << endl;
  cout << "    rmedge # # " << endl;
  cout << "    addnode #" << endl;
  cout << "    rmnode #" << endl;
  cout << "    isconn #" << endl;
  cout << "    adjo #" << endl;
  cout << "    adji #" << endl;
  cout << "    ptnodes" << endl;
  cout << "    ptedges" << endl;
  cout << "    cycle?" << endl;
  cout << "    topsort? #" << endl;
  cout << "    help" << endl;
  cout << "    quit" << endl;
  cout << endl;
}
```

Used in part 61.

7 Startup Script

"graph_proxy" 70b ≡

⟨graph_proxy graph proxy script 71a⟩
⟨graph_proxy graph proxy ns 71b⟩
⟨graph_proxy graph proxy imr 71c⟩
⟨graph_proxy graph proxy client 72a⟩

This script is meant to be run on UNIX systems with the Bourne shell. To invoke this script after running `nuweb` on `graph_proxy.w` you will need to type "`sh graph_proxy`". Four flags are supported: `-f` will run the first client presented in this paper, `-d` will cause the DII client to run, `-t` invokes the timing-test client, and `-i` runs the interactive client.

This script starts and kills the naming service after the client has been run once. The servers are run in `shared` mode; see pp. 34-35 of [Roemer99] for a discussion of the various server activation styles.

⟨graph_proxy graph proxy script 71a⟩ ≡

```
#!/bin/sh

PATH=../../daemon:../../coss/naming:../../imr:../../ir:$PATH
ADDR=inet:`uname -n`:12456
RC="-ORBImplRepoAddr $ADDR -ORBNamingAddr $ADDR"

echo Starting up Mico daemon
micod -ORBIIOPAddr $ADDR &
micod_pid=$!
sleep 1

trap "kill $micod_pid" 0
```

Used in part 70b.

The naming service must be "started up."

⟨graph_proxy graph proxy ns 71b⟩ ≡

```
echo "registering name service ..."
imr create NameService poa `which nsd` \
   IDL:omg.org/CosNaming/NamingContext:1.0#NameService $RC
```

Used in part 70b.

Next, the servers are registered and then activated.

⟨graph_proxy graph proxy imr 71c⟩ ≡

```
# register server
echo Starting up imr registration NS stuff
imr create mgraph_proxy_factory shared "`pwd`/mserver_ns $RC" \
    IDL:graph_proxy:1.0 $RC \

# register server
echo Starting up imr registration NS stuff
imr create agraph_proxy_factory shared "`pwd`/aserver_ns $RC" \
    IDL:graph_proxy:1.0 $RC \

echo Registering with NSD
imr activate mgraph_proxy_factory $RC
imr activate agraph_proxy_factory $RC

sleep 1
```

Used in part 70b.

The `sleep` command allows enough time for the servers to be activated before the client attempts to contact them.

⟨`graph_proxy` graph proxy client 72a⟩ ≡

```
case $1 in
  "-s")
    echo Starting server on $ADDR
    echo Contact with -ORBNamingAddr $ADDR
    sleep 9000
    ;;
  "-d")
    echo Starting: ./dclient_ns -ORBNamingAddr $ADDR
    ./dclient_ns -ORBNamingAddr $ADDR
    ;;
  "-i")
    echo Starting: ./iclient_ns -ORBNamingAddr $ADDR
    ./iclient_ns -ORBNamingAddr $ADDR
    ;;
  "-t")
    echo Starting: ./tclient_ns -ORBNamingAddr $ADDR
    ./tclient_ns -ORBNamingAddr $ADDR
    ;;
  "-f")
    echo Starting: ./client_ns -ORBNamingAddr $ADDR
    ./client_ns -ORBNamingAddr $ADDR
    ;;
  *)
    echo Must supply a client flag: -f, -d, -i or -t
esac
```

Used in part 70b.

8 Makefile

`"Makefile"` 72b ≡

⟨`Makefile` Makefile 73⟩

The following UNIX makefile provides for compilation of the programs presented in this paper, as well as for production of this formatted documentation.

⟨`Makefile` Makefile 73⟩ ≡

```
   ⟨Makefile vars 74a⟩

all:  servers clients

servers: a$(SERVER) m${SERVER}
clients: ${CLIENT} d$(CLIENT) t$(CLIENT) i$(CLIENT)

⟨Makefile server compilation 74b⟩
⟨Makefile client compilation 75a⟩
⟨Makefile ndisgraph compilation 75b⟩
⟨Makefile document production 75c⟩

$(PRJ).o: $(PRJ).cc
        mico-c++ $(CPPFLAGS) -I. -c $(PRJ).cc -o $(PRJ).o

$(PRJ).h $(PRJ).cc: $(PRJ).idl
        idl $(IDLFLAGS) $(PRJ).idl

$(PRJ).idl: $(PRJ).w
        nuweb $(PRJ).w
        chmod o+x $(PRJ)

#code-gen:
#       idl --use-dii --c++-impl ${IDLNAME).idl

clean:
        rm -fr *.o ?${SERVER} ?${CLIENT} ${CLIENT} *~ \
                *.tex *.log core *.idl *.inf *.toc\
                *.dvi *.ps *.aux *.cc *.h ${PRJ} \
                mdemo ademo
```

Used in part 72b.

Note that although [Roemer99], p. 15, states that use of the flag `--use-dii` has no affect on application implementation. However, in practice, we have found this not to be the case. If one intends to make DII calls from a client, the IDL must be run with `--use-dii`.

⟨`Makefile` vars 74a⟩ ≡

```
# For the distributed examples
VERSION = 2.2.5
PRJ = graph_proxy
SERVER = server_ns
CLIENT = client_ns
IDLFLAGS = --use-dii
CPPFLAGS = -g
MICLIB = -lmico$(VERSION) -lmicocoss$(VERSION)

# For the non distributed graph examples
BASEFILES=graph.h graph_interface.h
MATRIXFILES=matrixgraphbase.h matrixgraph.h \
main.cc $(BASEFILES)
ADJFILES=adjacencygraphbase.h adjacencygraph.h\
 main2.cc $(BASEFILES)
FLAGS=-ftemplate-depth-30
```

Used in part 73.

⟨`Makefile` server compilation 74b⟩ ≡

```
m$(SERVER): $(PRJ).o
        mico-c++ -DMAT $(CPPFLAGS) -I. \
-c $(SERVER).cc -o m$(SERVER).o
        mico-ld -I. $(PRJ).o m$(SERVER).o $(MICLIB) -o m$(SERVER)

a$(SERVER): $(PRJ).o
        mico-c++ -DADJ $(CPPFLAGS) -I. \
-c $(SERVER).cc -o a$(SERVER).o
        mico-ld -I. $(PRJ).o a$(SERVER).o $(MICLIB) -o a$(SERVER)
```

Used in part 73.

⟨`Makefile` client compilation 75a⟩ ≡

```
$(CLIENT): $(PRJ).o
        mico-c++ $(CPPFLAGS) -I. -c $(CLIENT).cc -o $(CLIENT).o
        mico-ld -I. $(PRJ).o $(CLIENT).o $(MICLIB) -o $(CLIENT)

d$(CLIENT): $(PRJ).o
        mico-c++ $(CPPFLAGS) -I. -c d$(CLIENT).cc -o d$(CLIENT).o
        mico-ld -I. $(PRJ).o d$(CLIENT).o $(MICLIB) -o d$(CLIENT)

i$(CLIENT): $(PRJ).o
        mico-c++ $(CPPFLAGS) -I. -c i$(CLIENT).cc -o i$(CLIENT).o
        mico-ld -I. $(PRJ).o i$(CLIENT).o $(MICLIB) -o i$(CLIENT)

t$(CLIENT): $(PRJ).o
        mico-c++ $(CPPFLAGS) -I. -c t$(CLIENT).cc -o t$(CLIENT).o
        mico-ld -I. $(PRJ).o t$(CLIENT).o $(MICLIB) -o t$(CLIENT)
```

Used in part 73.

The targets `mdemo` and `ademo` prompt compilation of the don-distributed graph tests.

⟨`Makefile` ndisgraph compilation 75b⟩ ≡

```
mdemo:  $(MATRIXFILES)
        mico-c++ main.cc $(FLAGS) -o mdemo

ademo:  $(ADJFILES)
        mico-c++ main2.cc $(FLAGS) -o ademo
```

Used in part 73.

The target `doc` produces this postscript document.

⟨`Makefile` document production 75c⟩ ≡

```
doc: $(PRJ).w
        nuweb $(PRJ).w; latex $(PRJ).tex
        nuweb $(PRJ).w; latex $(PRJ).tex
        dvips $(PRJ).dvi -o $(PRJ).ps
```

Used in part 73.

9 Limitations

Currently, CORBA has several limitations that hinder creation of a purely distributed environment. Many ORB vendors' proprietary ORB features hinder portability from one ORB to another. Furthermore, ORB interoperability is far from guaranteed.

Furthermore, we found production of a Java client to be impractical in the scope of this project due to the fact that use of the ORB provided by Sun Microsystems as part of Java SDK 2 requires storage and transmission of lengthy IORs for interoperability with the `MICO` ORB. Kay Roemer, co-creator of `MICO` has said, "...the JDK uses some proprietary mechanism to find initial references via those ORBInitial properties. The only portable way to make the JDK ORB use MICO's naming service is to pass the stringified IOR of mico's naming service to the Java application and turn it into an Object..."[Roemer99b]

10 Conclusion

In implementing our distributed graph system, we have found the use of proxy components to be an advantageous development approach. Proxy components enable high modularization and permit existing non-distributed systems to be migrated to a distributed setting without making changes to the source code of those existing systems.

Furthermore, we have found that a distributed scheme allows a user of a computation-intensive system like our graph system to get the results of computations more quickly. A server may be placed on a remote machine that is faster, or which has been otherwise optimized for the sort of computation to be performed. Even taking into account the overhead of network communication, significant time savings can then be realized, as shown by the results of our timing-test client runs.

11 Appendix A: Non-Distributed Graphs

In this appendix, we present two implementations of a generic STL graph container used in the system described in this paper. Our primary goal in the design of these containers was to facilitate end users' creation of graph classes with varying internal data structures, such as matrices and lists. For the purposes of demonstration, we have fully implemented a matrix graph and will discuss how to create other representations easily using this framework. Then, we present an adjacency-list graph representation which was built upon this framework.

The class hierarchy, in order from base class to the final graph class, is as follows: `Graph`, `GraphInterface`, followed by the implementation-specific classes: `MatrixGraphBase` and `matrix_graph` and the instantiable class. The implementation-specific files for our adjacency-list representation will be presented thereafter. Both implementations have been designed for compliance with the STL `Container` concept.

Other, established graph packages [Knuth93][GTL99] exist. Proxies similar to those developed in this document may be developed for such packages.

11.1 Graph Interface Reference

Listed here are all of the operations supported by both graph implementations. The label `GraphType` will, in the below listing, serves to represent, interchangeably, either a matrix graph or adjacency graph.

- `GraphType()`—Constructor; the graph is assumed to be undirected.
- `GraphType(bool isDigraph)`—Constructor, specifying via a boolean flag indicating whether the graph is directed.
- `GraphType( const GraphType& g )`—Copy constructor.
- `virtual void addNode ( const DataType& nodeName )`—Adds a node of the specified name.
- `void addNodes( ForwardIterator i1, ForwardIterator i2 )`— Adds a collection of nodes to the graph. The forward iterators passed in as parameters correspond to the locations in a container bounding a sequence of elements to be added to the graph.
- `Scalar addWeights ( ForwardIterator i1, ForwardIterator i2 ) const`— Sums the weights of the edges between a series of nodes of a specific order. The two parameters should point into a container of node names. If two or more consecutive nodes do not have an edge between them, no value is added to the total for the edge sequence.
- `void adjacentNodesIn ( const DataType& nodeName, OutputIterator o)`— Outputs, via the output iterator, all edges spanning into `nodeName`.

- `void adjacentNodesOut ( const DataType& nodeName, OutputIterator o)`— Outputs, via the output iterator, all edges spanning out of `nodeName`.
- `iterator begin()`—Returns an iterator to the first node in the node container.
- `const_iterator begin() const`—Returns a constant iterator to the first node in the node container.
- `void clear()`—Removes all nodes and edges.
- `bool empty() const`—Returns `true` if the container is empty (no nodes exist). Otherwise, `false` is returned.
- `iterator end()`—Returns an iterator to the end element in the node container.
- `const_iterator end()`—Returns a constant iterator to the end element in the node container.
- `virtual Scalar findWeight ( const DataType& nodeName1, const DataType& nodeName2 ) const`— Finds the weight of the edge between the two nodes given. If an edge does not exist between the two nodes, scalar zero is returned.
- `void getEdges ( OutputIterator o ) const`—Places all edges (node-name pairs) into an arbitrary container pointed to by the output iterator.
- `virtual bool isConnected(const DataType& nodeName) const` —Returns `true` if the specified node is not an orphan. Otherwise, it returns `false`.
- `void listAll( ostream& o ) const`—Fancy-prints all node pairs in the container to the output stream.
- `Scalar max_size() const`— Returns the maximum number of nodes in the node container.
- `virtual Scalar numEdges() const`—Returns the number of edges in the graph.
- `reverse_iterator rbegin()`—Returns a reverse iterator pointing to the last node in the node container.
- `const_reverse_iterator rbegin() const`—Returns a constant reverse iterator pointing to the last node in the node container.
- `reverse_iterator rend()`—Returns a reverse iterator pointing to the first node in the node container.

- `const_reverse_iterator rend() const`—Returns a constant reverse iterator pointing to the first node in the node container.

- `virtual void removeNode(const DataType& nodeName)`— Removes the node with the specified name.

- `void removeNodes(ForwardIterator i1, ForwardIterator i2)` — Removes a collection of nodes. The forward iterators passed in as parameters correspond to locations bounding a sequence of node names which are to be deleted from this container.

- `void removeAllNodes()`—Removes all nodes (and therefore all edges) from the graph. Has the exact same functionality as `clear()`.

- `virtual void removeOrphanNodes()`—This method removes all orphan nodes from the graph.

- `Scalar size() const`—Returns the size of (number of nodes in) the container.

- `virtual void swap(GraphType& g)`—Swaps the contents of two graph objects.

- `GraphType& operator=() const`—The assignment operator.

11.2 The Graph Class

The `Graph` class, the root class in the hierarchy, provides operations that *any* graph will need, regardless of implementation.

`Graph` maintains a unique mapping between node "names" and an internal scalar key. The node "names" are assigned by the end user— in the demonstration programs, we have chosen the name of be of type `string`, however, any type that supports the `operator==` operation should work in this context. The internal scalar key associates a scalar value with a any given node and is useful to some subclasses (such as the matrix) for indexing purposes.

It would probably be more efficient to dispose of the container which stores these pairings and simply handle any necessary mappings in the subclasses. However, this would put a greater burden on the designer of these implementation classes. The focus in the design of the base classes is more on modularity and less on maximizing efficiency. The idea is to make it easy for programmers to create new graph implementations using this framework; storing key/name pairs at the `Graph` level helps realize this goal.

The contents of `graph.h` are as follows:

"graph.h" 80a ≡

```
#ifndef GRAPH_H
#define GRAPH_H
⟨Graph directives, header files, and declarations 80b⟩
⟨The one_less_than adaptor definition 81a⟩
⟨The equal2nd adaptor definition 81b⟩
⟨Graph class definition and protected data members 82a⟩
⟨Graph constructors 82b⟩
⟨Other methods for the Graph class 84b⟩
⟨Graph code for adding and removing nodes 83⟩
⟨Information extraction methods for the Graph class 84a⟩
⟨Protected methods for use by subclasses of Graph 85⟩
⟨External overloaded output operations for Graph 86⟩
⟨graph_iterator class definition 87⟩
#endif
```

We begin by using a compiler directive to prevent recompilation of the header and include the necessary header files; we include such a directive in each of our source files. To avoid having to specify namespaces explicitly, we use the `using` construct to automatically assume the "std" namespace.

⟨`Graph` directives, header files, and declarations 80b⟩ ≡

```
#include <map>
#include <string>
#include <algorithm>
#include <pair.h>
#include <function.h>
#include <iostream.h>
#include <fstream>
#include <typeinfo>
#include "pairhelp.h"

using namespace std;

#define MAX_NODENAME_LENGTH     80
```

Used in part 80a.

The below adaptor, a `BinaryPredicate`, will be used with the STL `adjacent _find` algorithm. Its purpose is to assist in finding "gaps" in sequences of scalar indices. If nodes are added to a graph, these nodes automatically will be assigned sequentially increasing index values, starting with the first non-zero scalar value ("1" in the case of integers). If an element other than that at the end of the sequence is removed later on, it will, of course, no longer exist in the node-container.

By applying `adjacent_find` using the below adaptor, we can identify the first non-consecutive (non-existent) scalar value in the node container. The idea is to insert new nodes at the first "gap" to keep minimal the size of the graph data representation. (It is assumed that the container used for node pairings must exhibit an ordering of elements based on key (scalar) values).

Below, one can see that `operator()` is defined to check that the `first` element (the key) in a pair contains a value one less than the next in the ordered container. It will return `true` as long as this is the case and `false` otherwise, causing the `adjacent_find` algorithm to halt on the first non-consecutive element.

⟨The `one_less_than` adaptor definition 81a⟩ ≡

```
template <class Pair>
class one_less_than
{
public:
  typedef Pair first_argument_type;
  typedef Pair second_argument_type;
  typedef bool result_type;

  bool operator() (const Pair& p1, const Pair& p2) const {
    return (p2.first == p1.first+1);
  }
};
```

Used in part 80a.

The next adaptor is a `Predicate`. This adaptor simply allows for comparison of the value of a "name" element to the `second` element in a scalar/name pair.

⟨The `equal2nd` adaptor definition 81b⟩ ≡

```
template <class Pair, class CompareType>
class equal2nd
{
  private:
    CompareType equal_val;

  public:
    typedef Pair argument_type;

    equal2nd(CompareType t) : equal_val(t) {}
    bool operator() (const Pair& p) { return p.second==equal_val; }
};
```

Used in part 80a.

Now we come to the definition of the `Graph` class. Upon instantiation, the user must specify, via template parameters, the data types that will be used as "key" (must be a scalar type) and "name".

The protected section of the class contains a few type definitions to simplify naming. We revealed an egcs 1.0.3 compiler bug when attempting to compile one of the type definitions. To work around this problem, we have added support for changing this typedef to a macro definition if the `EGCS_TEMPLATE_BUG` macro is defined during compilation. If this macro is used, text substitutions are made by the preprocessor rather than during compilation itself.

After the type-definitions are two object declarations. One is for the ordered container (described previously) that stores scalar/data pairs. The second is a boolean flag indicating whether or not the given graph is directed.

⟨`Graph` class definition and protected data members 82a⟩ ≡

```
template<class KeyType, class DataType>
class Graph {

protected:
  typedef KeyType  Scalar;
  typedef pair<KeyType,DataType> GraphPair;

  #ifndef EGCS_TYPEDEF_BUG
  typedef map<KeyType,DataType,less<KeyType> > NodeContainer;
  #else
  #define NodeContainer map<KeyType,DataType,less<KeyType> >
  #endif

   NodeContainer nodeContainer;
   bool isDigraph;
```

Used in part 80a.

The `Graph` class supplies three constructors. The third is a copy constructor.

⟨`Graph` constructors 82b⟩ ≡

```
public:
  explicit Graph () : isDigraph(false)  { }

  explicit Graph (bool isDi) : isDigraph(isDi)  { }

  Graph ( const Graph& g ) : nodeContainer(g.nodeContainer),
                             isDigraph(g.isDigraph) {}
```

Used in part 80a.

The `addNode()` method adds a node name to the container and automatically assigns it an internal scalar index, starting with 1. The scalar value and node name are kept in an STL `pair`. The `adjacent_find` algorithm, in conjunction with the `one_less_than` adaptor (described above), finds gaps in index numbers within the container and assigns a new node to a "gap" index, if one exists. Subclasses must override this method by first calling this version of the method and then performing any implementation-specific bookkeeping that may be necessary.

The `removeNode` method removes an existing node-pair from the container. Like `addNode`, subclasses must override this method by first calling this version and then performing any implementation-specific tasks.

⟨Graph code for adding and removing nodes 83⟩ ≡

```
virtual void addNode( const DataType& nodeName )  {
  NodeContainer::iterator i;

  if (nodeContainer.empty())
    nodeContainer[1]=nodeName;

  else if((i=adjacent_find(nodeContainer.begin(),
    nodeContainer.end(),
    not2( one_less_than<GraphPair>()))) !=
          nodeContainer.end())
    nodeContainer[(*i).first+1] = nodeName;
  else nodeContainer[nodeContainer.size()+1] = nodeName;
}

virtual void removeNode ( const DataType& nodeName )  {

  NodeContainer::iterator i =
      find_if ( nodeContainer.begin(),
                nodeContainer.end(),
                equal2nd<GraphPair,DataType>(nodeName) );
 if (i != nodeContainer.end())
  nodeContainer.erase(i);
}

virtual void removeAllNodes()  {  nodeContainer.clear(); }
```

Used in part 80a.

The Graph class provides only a few methods for node-information extraction—the size, getIndexOf, and listAll methods, which are all constant. size returns the number of nodes in the graph; getIndexOf returns the index of a given node-data item (this information has proved indispensable in algorithm implementation). listAll fancy-prints all node pairings to an ostream.

⟨Information extraction methods for the `Graph` class 84a⟩ ≡

```
virtual Scalar size() const { return nodeContainer.size(); }

virtual Scalar getIndexOf ( const DataType& nodeName ) const {
  NodeContainer::const_iterator i =
    find_if( nodeContainer.begin(), nodeContainer.end(),
             pair_second_equal_to<DataType>(nodeName));

  Scalar scale(0);
  while (i-- != nodeContainer.begin())
        scale++;
  return scale;
}

virtual void listAll(ostream& o) const {
  NodeContainer::const_iterator first=nodeContainer.begin(),
                                last=nodeContainer.end();

  o << string("Scalar/NodeName Node Pairs for this ") +
       (isDigraph ? "directed" : "undirected") << " graph:"
       << endl;

  while (first != last)
  {   o << "(" << (*first).first << ", " <<
           (*first).second << ")" << endl; first++;
  }
}
```

Used in part 80a.

The below operations are required by the STL `Container`concept:

⟨Other methods for the `Graph` class 84b⟩ ≡

```
Graph& operator=( const Graph& g ) {
  if ( &g == this )
    return *this;

  nodeContainer = g.nodeContainer;
  isDigraph = g.isDigraph;
  return *this;
}

virtual void swap ( Graph& g ) {
  nodeContainer.swap( g.nodeContainer );
  bool tmp = isDigraph;
  isDigraph = g.isDigraph;
  g.isDigraph = tmp;
}
```

Used in part 80a.

The following methods are supplied for use by subclasses.

The `nodeExists()` method determines whether a node of a specified "name" exists in the graph and returns the truth value of its existence as a `bool`.

The `getScalarOf()` and `getIndexOf()` methods are similar but have one subtle difference: The former returns the actual key value of the pair in the container for a given node. The latter returns an entry from the consecutive numbering of nodes in the container. For example, if the third element in the container had a key value of 7, `getScalarOf()` would return 7, whereas `getIndexOf()` would return 2 (meaning the third ordered element, assuming 0 is a valid index).

The `nodeNameByIndex()` retrieves the "name" of a node based on an index number. A precondition to this method is that the index is valid.

⟨Protected methods for use by subclasses of `Graph` 85⟩ ≡

```
protected:
  bool nodeExists ( const DataType& nodeName ) const {
    return (find_if( nodeContainer.begin(), nodeContainer.end(),
      pair_second_equal_to<DataType>(nodeName)) !=
                                nodeContainer.end());
  }

  // Precondition- nodeName is a *valid* node name.
  Scalar getScalarOf ( const DataType& nodeName ) const {
    return (*(find_if( nodeContainer.begin(), nodeContainer.end(),
             pair_second_equal_to<DataType>(nodeName)))).first;
  }

  DataType nodeNameByIndex ( Scalar index ) const {
    NodeContainer::const_iterator i = nodeContainer.begin();
    Scalar x(0);
    while ( x < index ) { x++; i++; }
    return (*i).second;
  }
};
```

Used in part 80a.

The following output operation calls the `listAll()` method described above for any `ostream` object. It is declared globally because the compiler must distinguish between types on a global scope when determining which version of `operator<<` to execute during scope resolution.

⟨External overloaded output operations for `Graph` 86⟩ ≡

```
template <class KeyType, class DataType>
ostream& operator<< (ostream& o, const Graph<KeyType,DataType>& g)
{
  g.listAll(o);
  return o;
}
```

Used in part 80a.

The `graph_iterator`, defined below, is provided for use via the type-definitions `iterator`, `reverse_iterator`, `const_iterator` and `const_reverse_iterator`. The following iterator class is actually an adaptor class for all of the iterators supported by the node container. It also models the `ForwardIterator` concept. The entire class definition is as follows:

⟨graph_iterator class definition 87⟩ ≡

```
template <class Iterator, class DataType>
class graph_iterator
{
private:
  Iterator i;

public:
  typedef bidirectional_iterator_tag  iterator_category;
  typedef ptrdiff_t                   difference_type;
  typedef DataType                    value_type;
  typedef value_type&                 reference;
  typedef value_type*                 pointer;
  typedef graph_iterator              self;
  typedef self&                       self_ref;

  graph_iterator() {}

  graph_iterator( Iterator j ) : i(j) {}

  self  operator++(int) {
    self tmp = *this;
    ++(*this);
    return tmp;
  }

  self& operator++() { i++; return (*this); }
  self& operator--() { i--; return (*this); }
  self  operator--(int) {
    self tmp = *this;
    *this--;
    return tmp;
  }

  bool operator==(const graph_iterator& it) const
  { return (i== it.i); }

  bool operator!=(const graph_iterator& it) const
  { return (i != it.i); }

  reference operator*()
  { return (*i).second;  }

  pointer operator->()
  { return &(*i).second; }
};
```

Used in part 80a.

11.3 The GraphInterface Class

The `GraphInterface` class defines all methods that need to be supported by any implementation. Most methods in this class are therefore abstract, making it necessary for subclasses to implement them for successful compilation. End users of graph implementations will find the available primitive graph operations (services) here.

The contents of the `graph_interface.h` file are as follows:

"graph_interface.h" 88a ≡
```
#ifndef GRAPH_INTERFACE_H
#define GRAPH_INTERFACE_H
⟨GraphInterface directives and header files 88b⟩
⟨Class definition and constructors for GraphInterface 88c⟩
⟨GraphInterface public interface 90⟩
#endif
```

Here are the directives and header files for the `GraphInterface` class:

⟨`GraphInterface` directives and header files 88b⟩ ≡
```
#include "graph.h"
#include <vector>
```

Used in part 88a.

The `GraphInterface` class supports constructors analogous to those defined in the `Graph` class. No additional is done at this level—the parent constructors are simply invoked from here.

⟨Class definition and constructors for `GraphInterface` 88c⟩ ≡
```
template <class KeyType,class DataType>
class GraphInterface : public Graph<KeyType,DataType>
{
public:
  typedef KeyType Scalar;

  explicit GraphInterface() : Graph() {}

  explicit GraphInterface(bool isDi) : Graph(isDi) {}

  GraphInterface( const GraphInterface& g ) : Graph(g) {}
```

Used in part 88a.

These are the new public interface methods, in addition to those few defined in `Graph`. Again, most are abstract; those that are not are those that cannot be due to the use of templates.

The `AdjacentNodesIn()` and `AdjacentNodesOut()` methods both return (via an `OutputIterator`) a sequence of nodes which have edges leading into or out of, respectively, the specified node.

The `isConnected()` method determines whether a given node is connected by an edge to any other nodes in the graph.

`findWeight()` returns the weight of an edge between two specified nodes.

`addWeights()` sums the weights between nodes in an ordered list. A precondition to this method is that edges exist between the nodes.

`removeOrphanNodes()` removes nodes that are not connected to any other node in the graph.

`removeNodes()` is written in terms of `removeNode()` and allows the user to eliminate several nodes in one method call using iterators that point into an arbitrary container of node names.

`addEdge()` and `removeEdge()`, as the names imply, add and remove edges from the graph. These methods are purely implementation-specific, as there is no notion of an "edge" in the `Graph` class.

`addEdges()` adds a series of edges to a graph. The forward iterators specified as arguments should point into a container of node-name pairs.

`getEdges()` outputs all edges in the graph as node-name pairs into using an output iterator.

⟨GraphInterface public interface 90⟩ ≡

```
template <class OutputIterator>
bool adjacentNodesOut(const DataType& nodeName,
                             OutputIterator o) const;

template <class OutputIterator>
bool adjacentNodesIn(const DataType& nodeName,
                             OutputIterator o) const;

virtual bool isConnected(const DataType& nodeName) const=0;

virtual Scalar findWeight ( const DataType& nodeName1,
                             const DataType& nodeName2 ) const=0;


template < class ForwardIterator >
Scalar addWeights ( ForwardIterator iterator1,
                    ForwardIterator iterator2 ) const;

virtual void removeOrphanNodes()=0;

template <class ForwardIterator>
void removeNodes ( ForwardIterator iterator1,
                   ForwardIterator iterator2 );

virtual bool addEdge(const DataType& nodeName1,
                     const DataType& nodeName2,
                     const Scalar weight)=0;

template <class ForwardIterator>
bool addEdges( ForwardIterator begin, ForwardIterator end );

virtual bool removeEdge(const DataType& nodeName1,
                        const DataType nodeName2)=0;

template <class OutputIterator>
void getEdges ( OutputIterator o );
};
```

Used in part 88a.

11.4 The MatrixGraphBase Class

Up until this point, we have given source and documentation for the underlying framework of any graph implementation. However, a user wishing to create a class implementing a graph in a particular way will derive that implementation from `GraphInterface` and provide the required functionality.

This section presents an implementation of one particular type of graph class—the matrix-based graph. We will briefly discuss issues pertaining to the successful derivation of other user-defined graph classes from those already given, with the intention of making creation other implementations rather simple.

Other types of graph classes need not conform to the programming style of this implementation, as long as all required methods are implemented (correctly).

The `matrixgraphbase.h` file, containing the matrix-based graph implementation and supporting classes, contains the following:

```
"matrixgraphbase.h" 91a ≡
    #ifndef MATRIX_GRAPH_BASE_H
    #define MATRIX_GRAPH_BASE_H
    ⟨MatrixGraphBase directives and header files 91b⟩
    ⟨The Edge class definition 92⟩
    ⟨MatrixGraphBase class definition and private data members 93a⟩
    ⟨MatrixGraphBase constructors and destructors 93b⟩
    ⟨Container methods for MatrixGraphBase 102a⟩
    ⟨MatrixGraphBase interface methods 95a⟩
    ⟨Protected methods for MatrixGraphBase 94⟩
    ⟨External overloaded output operations for MatrixGraphBase 102b⟩
    #endif
```

Here are the directives and header-file inclusion for `matrixgraphbase.h`:

```
⟨MatrixGraphBase directives and header files 91b⟩ ≡
    #include "graph_interface.h"
```

Used in part 91a.

The following class represents an edge. In a matrix-based implementation, every matrix entry consists of an `Edge` object. The private attributes of the `Edge` object consist of the "edge" flag, specifying whether or not the edge exists, as well as a "hidden" flag, which allows an existing edge to be ignored. There also exists a "weight" attribute holding a generic scalar value and represents the weight of the given edge.

Most of the methods for this class are straightforward. However, we would like to point out a subtle difference between the methods `edgeExists()` and `isEdge()`. The former method specifies whether an edge exists whether it is hidden or not—that is, it ignores the hidden attribute of the edge. The latter specifies that the edge exists only if it is both an edge and is not hidden.

⟨The `Edge` class definition 92⟩ ≡

```
template< class Scalar >
class Edge
{
private:
  bool edge, hidden;
  Scalar weight;

public:
  Edge() : hidden(false), edge(false), weight(0) { }

  void makeEdge( Scalar scale ) { edge=true; weight=scale; }

  void removeEdge() { edge=false; }

  bool isEdge() const { return (edge && !hidden); }

  bool edgeExists() const { return edge; }

  Scalar getWeight() const {  return weight; }

  void setWeight(Scalar wgt) { weight=wgt; }

  bool isHidden() const { return hidden; }

  bool hide() { if (isEdge())
                 return false;
                else hidden=true;
                return true;    }

  bool unhide() { if (!isHidden())
                   return false;
                  else
                    hidden=false;
                  return true;    }
};
```

Used in part 91a.

Now we present the class definition, type-definitions and private data members for the `MatrixGraphBase` class. The template parameters are the same as those for `GraphInterface`. The `MatrixRows` container is a two-dimensional vector, where a vector of `MatrixColumns` are the elements of each `MatrixRows` index. STL vectors allow the indexing scheme used to access matrix elements to be the same as if it were done using primitive C++ multidimensional arrays.

⟨MatrixGraphBase class definition and private data members 93a⟩ ≡

```
template<class KeyType,class DataType>
class MatrixGraphBase :
  public GraphInterface<KeyType,DataType> {
  private:
    typedef KeyType Scalar;
    typedef Edge<Scalar> EdgeType;
    typedef vector<EdgeType> MatrixColumns;
    typedef vector<MatrixColumns> MatrixRows;

    MatrixRows       rows;
    NodeContainer    tempContainer;
```

Used in part 91a.

These are the constructors for the `MatrixGraphBase` class. Again, they have the same signatures and semantics as the constructors in the `GraphInterface` and `Graph` classes.

⟨MatrixGraphBase constructors and destructors 93b⟩ ≡

```
public:
  explicit MatrixGraphBase() :
           GraphInterface() {}


  explicit MatrixGraphBase(bool isDi) :
           GraphInterface(isDi) {}

  MatrixGraphBase ( const MatrixGraphBase& g ) :
        GraphInterface(g), rows(b.rows), tempContainer() {}

  ~MatrixGraphBase() {}
```

Used in part 91a.

The below protected methods are for internal use by this class and its subclasses. In this particular application, these methods aren't used by subclasses, but are available in case the developer decides he needs them. The allocation methods, `allocateGraph()`, `deAllocateGraph()` and `reallocateGraph` are used for initializing the matrix to the appropriate size (`allocateGraph()`), deleting the matrix (`deAllocateGraph()`) or restructuring an existing matrix (`reAllocateGraph()`. The `reAllocateGraph()` method is called whenever a node is deleted from the graph. Because the node has been deleted, any prior references to the node must be eliminated and the appropriate rows and columns updated within the matrix.

⟨Protected methods for `MatrixGraphBase` 94⟩ ≡

```
    void allocateGraph() {
    if (nodeContainer.size()>0)
      for (Scalar x(0); x < nodeContainer.size();x++)
        rows.push_back( MatrixColumns(nodeContainer.size()));
    }

    void deAllocateGraph() {
      rows.clear();
    }

    void reAllocateGraph() {
      if (!rows.empty())
      {
        NodeContainer::iterator it;
        pair<NodeContainer::iterator, NodeContainer::iterator>
          matchPair =
             mismatch ( tempContainer.begin(),
                        tempContainer.end(),
                        nodeContainer.begin());
        // Nodes were added
        while (tempContainer.size() < nodeContainer.size()) {
          Scalar count(0);
          it = matchPair.second;
          while (it-- != nodeContainer.begin()) count++;

          tempContainer.insert( *matchPair.second );
          rows.insert ( rows.begin()+count,
                        MatrixColumns(tempContainer.size()));
          for (Scalar y(0); y < rows.size(); ++y)
            rows[y].insert(rows[y].begin()+count, EdgeType());
        }
        // Nodes were removed
        while (tempContainer.size() > nodeContainer.size()) {
          Scalar count(0);
          it = matchPair.first;

          while (it-- != tempContainer.begin()) count++;

          tempContainer.erase(matchPair.first);
          rows.erase( rows.begin()+count );
          for (Scalar y(0); y < rows.size(); y++)
            rows[y].erase( rows[y].begin()+count );
        }
      }
      tempContainer.clear();
    }
  };
```

Used in part 91a.

The following methods implement the abstract methods in the `GraphInterface` class. For a brief explanation of the purpose of each method, please see the documentation for the `GraphInterface` class above.

⟨`MatrixGraphBase` interface methods 95a⟩ ≡

> ⟨Matrix implementation of `adjacentNodesOut` 95b⟩
> ⟨Matrix implementation of `adjacentNodesIn` 96a⟩
> ⟨Matrix implementation of `isConnected` 96b⟩
> ⟨Matrix implementation of `findWeight` 97a⟩
> ⟨Matrix implementation of `addWeights` 97b⟩
> ⟨Matrix implementation of `removeOrphanNodes` 97c⟩
> ⟨Matrix implementation of `addNode` 98a⟩
> ⟨Matrix implementation of `addNodes` 98b⟩
> ⟨Matrix implementation of `removeNode` 98c⟩
> ⟨Matrix implementation of `removeNodes` 98d⟩
> ⟨Matrix implementation of `removeAllNodes` 99a⟩
> ⟨Matrix implementation of `addEdge` 99b⟩
> ⟨Matrix implementation of `addEdges` 99c⟩
> ⟨Matrix implementation of `removeEdge` 100a⟩
> ⟨Matrix implementation of `getEdges` 100b⟩
> ⟨Matrix implementation of `numEdges` 101a⟩
> ⟨Matrix implementation of `listAll` 101b⟩

Used in part 91a.

⟨Matrix implementation of `adjacentNodesOut` 95b⟩ ≡

```
template  <class OutputIterator>
bool adjacentNodesOut(const DataType& nodeName,
                      OutputIterator o) const {
  if( numEdges() == (Scalar)(0) )
    return false;
  if (!nodeExists(nodeName))
    return false;
  Scalar index = getIndexOf(nodeName);
  const MatrixColumns& columns = rows[index];
  NodeContainer::const_iterator nodeIterator=
    nodeContainer.begin();
  MatrixColumns::const_iterator matrixIterator=columns.begin();

  for (Scalar x(0); matrixIterator != columns.end();
                    matrixIterator++, nodeIterator++, x++)
    if ( (*matrixIterator).isEdge() && x != index )
      *o = (*nodeIterator).second;

  return true;
}
```

Used in part 95a.

⟨Matrix implementation of `adjacentNodesIn` 96a⟩ ≡

```
template <class OutputIterator>
bool adjacentNodesIn(const DataType& nodeName,
                            OutputIterator o) const {
  if (!nodeExists(nodeName))
    return false;
  Scalar index = getIndexOf(nodeName);
  NodeContainer::const_iterator nodeIterator=
    nodeContainer.begin();

  for (Scalar x(0); x < rows.size(); x++, nodeIterator++) {
    if (x==index) // Skip the row corresponding to "nodeName"
      continue;

    if (rows[x][index].isEdge())
      *o = (*nodeIterator).second;
  }
  return true;
}
```

Used in part 95a.

⟨Matrix implementation of `isConnected` 96b⟩ ≡

```
virtual bool isConnected(const DataType& nodeName) const {
  if( numEdges() == (Scalar)(0) )
    return false;
  if (!nodeExists(nodeName))
    return false;
  Scalar index = getIndexOf(nodeName);
  const MatrixColumns& columns = rows[index];

  for ( Scalar x(0); x < columns.size(); x++ )
    if (columns[x].isEdge() && x != index) return true;

  for ( Scalar x(0); x < rows.size(); x++)
    if (rows[x][index].isEdge() && x != index) return true;

  return false;
}
```

Used in part 95a.

⟨Matrix implementation of `findWeight` 97a⟩ ≡

```
virtual Scalar findWeight ( const DataType& nodeName1,
                            const DataType& nodeName2 ) const {
  if (!nodeExists(nodeName1) | !nodeExists(nodeName2))
    return Scalar(0);

  return rows[getIndexOf(nodeName1)]
             [getIndexOf(nodeName2)].getWeight();
}
```

Used in part 95a.

⟨Matrix implementation of `addWeights` 97b⟩ ≡

```
template < class ForwardIterator >
 Scalar addWeights ( ForwardIterator iterator1,
                     ForwardIterator iterator2 ) const {
   Scalar scale(0);
   ForwardIterator tmp=iterator1;

   while (++iterator1 != iterator2)
     scale = scale + findWeight( *tmp++, *iterator1 );

   return scale;
 }
```

Used in parts 95a, 112.

⟨Matrix implementation of `removeOrphanNodes` 97c⟩ ≡

```
virtual void removeOrphanNodes()  {
  vector<DataType> removeContainer;

  for ( NodeContainer::iterator i=nodeContainer.begin();
                               i != nodeContainer.end(); i++ )
  {
    if (!isConnected( (*i).second ))
      removeContainer.push_back( (*i).second );
  }
  removeNodes ( removeContainer.begin(), removeContainer.end() );
}
```

Used in parts 95a, 112.

⟨Matrix implementation of `addNode` 98a⟩ ≡

```
virtual void addNode ( const DataType& nodeName ) {
  copy ( nodeContainer.begin(), nodeContainer.end(),
         inserter(tempContainer, tempContainer.begin()));

  Graph::addNode(nodeName);
  reAllocateGraph();
}
```

Used in part 95a.

⟨Matrix implementation of `addNodes` 98b⟩ ≡

```
template <class ForwardIterator>
void addNodes ( ForwardIterator iterator1,
                ForwardIterator iterator2 ) {
  while (iterator1 != iterator2)
  { addNode(*iterator1); iterator1++; }
}
```

Used in parts 95a, 112.

⟨Matrix implementation of `removeNode` 98c⟩ ≡

```
virtual void removeNode ( const DataType& nodeName ) {
  copy ( nodeContainer.begin(), nodeContainer.end(),
  inserter(tempContainer, tempContainer.begin()));

  Graph::removeNode(nodeName);
  reAllocateGraph();
}
```

Used in part 95a.

⟨Matrix implementation of `removeNodes` 98d⟩ ≡

```
template <class ForwardIterator>
  void removeNodes ( ForwardIterator iterator1,
                     ForwardIterator iterator2 )
  {
       while ( iterator1 != iterator2 ) {
          removeNode ( *iterator1 );
          iterator1++;
        }
   }
```

Used in parts 95a, 112.

⟨Matrix implementation of `removeAllNodes` 99a⟩ ≡

```
virtual void removeAllNodes() {
  Graph::removeAllNodes();
  deAllocateGraph();
}
```

Used in part 95a.

⟨Matrix implementation of `addEdge` 99b⟩ ≡

```
virtual bool addEdge(const DataType& nodeName1,
                     const DataType& nodeName2,
                     const Scalar weight=0)
{
  if (!nodeExists(nodeName1) || !nodeExists(nodeName2))
    return false;

  if (rows.empty())
    allocateGraph();

  Scalar index1 = getIndexOf(nodeName1);
  Scalar index2 = getIndexOf(nodeName2);

  rows[index1][index2].makeEdge(weight);

  if (!isDigraph && nodeName1 != nodeName2 )
    rows[index2][index1].makeEdge(weight);

  return true;
}
```

Used in part 95a.

⟨Matrix implementation of `addEdges` 99c⟩ ≡

```
template <class ForwardIterator>
bool addEdges ( ForwardIterator begin, ForwardIterator end ) {
  for (; begin != end;  begin++)
    if (!addEdge( (*begin).first, (*begin).second ))
      return false;
  return true;
}
```

Used in parts 95a, 112.

⟨Matrix implementation of `removeEdge` 100a⟩ ≡

```
virtual bool removeEdge(const DataType& nodeName1,
                        const DataType nodeName2)
{
  if (!nodeExists(nodeName1) || !nodeExists(nodeName2))
    return false;

  Scalar index1 = getIndexOf(nodeName1);
  Scalar index2 = getIndexOf(nodeName2);

  rows[index1][index2].removeEdge();

  if (!isDigraph && nodeName1 != nodeName2 )
    rows[index2][index1].removeEdge();

  return true;
}
```

Used in part 95a.

⟨Matrix implementation of `getEdges` 100b⟩ ≡

```
template <class OutputIterator>
void getEdges ( OutputIterator o ) const {
  for (Scalar x(0); x < rows.size(); x++)
    for (Scalar y(0); y < rows[x].size(); y++)
    {
      if (x == y) continue;
      if (rows[x][y].isEdge())
      {
        const pair<DataType,DataType> p(nodeNameByIndex(x),
                                        nodeNameByIndex(y));
        *o = p;
      }
   }
}
```

Used in part 95a.

⟨Matrix implementation of `numEdges` 101a⟩ ≡

```
virtual Scalar numEdges() const {
  Scalar count=0;

  for (Scalar x(0); x < rows.size();x++)
    for (Scalar y(0); y < rows[x].size(); y++)
      if (rows[x][y].isEdge() && x != y )
        count++;

  if (!isDigraph)
    count /= 2;
  return count;
}
```

Used in part 95a.

⟨Matrix implementation of `listAll` 101b⟩ ≡

```
virtual void listAll(ostream& o) const {
  Graph::listAll(o);

  o << endl << "Edge listings: ";
  if (rows.size()) o << endl;
  else o << "None." << endl;

  MatrixColumns::const_iterator mci;
  MatrixRows::const_iterator mri;
  NodeContainer::const_iterator nciX, nciY;

  for (mri=rows.begin(), nciX=nodeContainer.begin();
                         mri != rows.end(); mri++, nciX++)
    for (mci= (*mri).begin(), nciY=nodeContainer.begin();
                              mci != (*mri).end(); mci++, nciY++)
    {
      if ( (*mci).isEdge())
        cout << (*nciX).second << " has an edge to " <<
                (*nciY).second << " of weight: " <<
                (*mci).getWeight() << endl;
    }
}
```

Used in part 95a.

The following methods are required for compliance with the STL `Container` concept. They perform the necessary work specific to matrix-graphs and also call the `Graph` version of each.

⟨Container methods for `MatrixGraphBase` 102a⟩ ≡

```
public:
  MatrixGraphBase& operator= ( const MatrixGraphBase& g ) {
    if ( &g == this )
      return *this;

    Graph::operator=(g);

    rows = g.rows;
    tempContainer.clear();
    return *this;
  }

  virtual void swap ( MatrixGraphBase& g ) {
    Graph::swap(g);
    rows.swap(g.rows);
    tempContainer.swap(g.tempContainer);
  }
```

Used in part 91a.

Here is an overloaded output operator specific to the `MatrixGraphBase` class:

⟨External overloaded output operations for `MatrixGraphBase` 102b⟩ ≡

```
template<class KeyType, class DataType>
ostream& operator<< (ostream& o,
                     const MatrixGraphBase<KeyType,DataType>& g)
{
  g.listAll(o);
  return o;
}
```

Used in part 91a.

11.5 The Instantiable Matrix-Based Graph

The `matrix_graph` class resides in the file `matrixgraph.h` and is the lowest (instantiable) class in the hierarchy. This class conforms to the STL `Forward-Container` and `ReversibleContainer` concepts. Among these requirements is an iterator for traversing the nodes within the container; and all other methods required by `Container` are also given.

An outline of the `matrixgraph.h` file is as follows:

"matrixgraph.h" 103a ≡

```
#ifndef MATRIX_GRAPH_H
#define MATRIX_GRAPH_H
⟨matrix_graph directives and header files 103b⟩
⟨matrix_graph class definition and type-definitions 103c⟩
⟨Constructors for matrix_graph 104a⟩
⟨matrix_graph container wrappers for STL compliance 104b⟩
#endif
```

Once again, the directives and header file:

⟨`matrix_graph` directives and header files 103b⟩ ≡

```
#include "matrixgraphbase.h"
```

Used in part 103a.

The `matrix_graph` class, defined below, is the final product to be manipulated by the end user and is suitable for use with any appropriate STL algorithm. The template parameters are the same as those for `MatrixGraphBase` and `GraphInterface`. The type-definitions required by an STL `ForwardContainer` and `ReversibleContainer` are also given:

⟨`matrix_graph` class definition and type-definitions 103c⟩ ≡

```
template <class KeyType, class DataType>
class matrix_graph :
  public MatrixGraphBase<KeyType,DataType>
{
public:
  typedef DataType                        value_type;
  typedef DataType&                       reference_type;
  typedef const DataType&                 const_reference;
  typedef DataType*                       pointer;
  typedef ptrdiff_t                       difference_type;
  typedef size_t                          size_type;
  typedef graph_iterator
    <NodeContainer::iterator,DataType> iterator;
  typedef graph_iterator
    <NodeContainer::const_iterator,DataType> const_iterator;
  typedef graph_iterator
    <NodeContainer::reverse_iterator,DataType>
       reverse_iterator;
  typedef graph_iterator
    <NodeContainer::const_reverse_iterator,DataType>
      const_reverse_iterator;
```

Used in part 103a.

Listed below are the constructors, which follow the same semantics as the constructors for the previous classes.

⟨Constructors for `matrix_graph` 104a⟩ ≡

```
explicit matrix_graph( bool isDirected ) :
                       MatrixGraphBase(isDirected) {}

explicit matrix_graph () : MatrixGraphBase() {}

matrix_graph ( const matrix_graph& g ) :
                     MatrixGraphBase(g) { }
```

Used in part 103a.

Here are the methods required for STL compliance. They are, more or less, wrappers for the node container's methods, since it is the container that the user will actually be accessing. Other methods merely call the original method from a base class.

⟨`matrix_graph` container wrappers for STL compliance 104b⟩ ≡

```
  iterator begin() { return nodeContainer.begin(); }
  const_iterator begin() const { return nodeContainer.begin(); }
  iterator end() { return nodeContainer.end(); }
  const_iterator end() const { return nodeContainer.end(); }
  reverse_iterator rbegin() { return nodeContainer.rbegin(); }
  const_reverse_iterator rbegin() const
    { return nodeContainer.rbegin(); }
  reverse_iterator rend() { return nodeContainer.rend(); }
  const_reverse_iterator rend() const
    { return nodeContainer.rend(); }
  bool empty() const { return nodeContainer.empty(); }
  void clear() { removeAllNodes(); deAllocateGraph(); return; }
  void swap ( matrix_graph& g ) { GraphInterface::swap(g); }
  Scalar max_size() const { return nodeContainer.max_size(); }

  matrix_graph& operator=( const matrix_graph& g ) {
    MatrixGraphBase::operator=(g);
    return *this;
  }
};
```

Used in part 103a.

11.6 The pairhelp.h Header File

The `pairhelp.h` header file contains methods for testing equality between an arbitrary element and the first or second elements of a pair. They do not contain any code specific to graphs and therefore may be reused in other applications that may require access to a pair element for purposes of testing equality.

"pairhelp.h" 105a ≡
```
#ifndef PAIRHELP_H
#define PAIRHELP_H
⟨pair_first_equal_to class definition 105b⟩
⟨pair_second_equal_to class definition 105c⟩
#endif
```

Class `pair_first_equal_to` is a `Predicate` which accepts an arbitrary type via its constructor that must be the same as the type of the `first` pair element. Later, when the `operator()` method is called, a `pair` is passed in as an argument and compared to the element passed in during construction. The operator returns `true` if the elements are equal and `false` otherwise.

⟨pair_first_equal_to class definition 105b⟩ ≡
```
template <class ConstructorType>
class pair_first_equal_to
{
private:
  const ConstructorType& ct;
public:
  pair_first_equal_to ( const ConstructorType& t ) : ct(t) {}

  template <class Type>
  bool operator() ( const Type& t ) { return (t.first == ct); }
};
```

Used in part 105a.

The second `Predicate` is the same as the first, except that the `second` pair element is the one checked for equality against the item passed in during construction:

⟨pair_second_equal_to class definition 105c⟩ ≡
```
template <class ConstructorType>
class pair_second_equal_to
{
private:
  const ConstructorType& ct;

public:
  pair_second_equal_to ( const ConstructorType& t  ) : ct(t) {}

  template <class Type>
  bool operator() ( const Type& t ) { return (t.second == ct); }
 };
```

Used in part 105a.

11.7 Matrix Graph Test Application

The below test application demonstrates some of the capabilities of the `matrix_graph` class.

"main.cc" 106a ≡
⟨Header files for the main program 106b⟩
⟨Declare a `matrix_graph` object 106c⟩
⟨First test 107a⟩
⟨Second test 107b⟩
⟨Third test 108a⟩
⟨Fourth test 108b⟩
⟨Fifth test 109a⟩

First, we include the header file for the `matrix_graph` class:

⟨Header files for the main program 106b⟩ ≡

```
#include "matrixgraph.h"
```

Used in part 106a.

Then, we declare the graph object we will use.

⟨Declare a `matrix_graph` object 106c⟩ ≡

```
int main()
{
  matrix_graph<unsigned int,string> g(true);
```

Used in part 106a.

We start by adding four nodes and demonstrating how to iterate through our graph container. After that, we add a couple of edges, demonstrate how output the contents (nodes) of the graph, and finally show how large the container is using `size()`.

⟨First test 107a⟩ ≡

```
g.addNode("New York");
g.addNode("Fort Lauderdale");
g.addNode("Los Angeles");
g.addNode("Ridgefield");

cout << "A listing of all nodes using the graph iterator: ";
copy ( g.begin(), g.end(),
       ostream_iterator<string>(cout, "\n"));
cout << endl << endl;

g.addEdge("New York", "Los Angeles", 3000);
g.addEdge("Fort Lauderdale", "New York", 1500);

cout << g << endl;
cout << "There are " << g.size()
     << " nodes in the graph" << endl;
```

Used in parts 106a, 122b.

Next, we remove a node and add a different node and a new edge. This demonstrates how a "gap" in the node sequence is left by the removal of the Fort Lauderdale node and also to show that any edges leading to or from Fort Lauderdale were removed.

⟨Second test 107b⟩ ≡

```
g.removeNode("Fort Lauderdale");
g.addNode("Nowheresville");
g.addEdge("New York", "Nowheresville", 50);
cout << g;
```

Used in parts 106a, 122b.

Next, we test the `adjacentNodesOut()` and `adjacentNodesIn()`. Note that an arbitrary container can be used to store the results of these two calls—for this demonstration, we arbitrarily chose a `vector`.

After demonstrating those two functions, we proceed to show that Ridgefield is not connected to any other node by using the `isConnected()` method, and finally adding an edge to Ridgefield from Los Angeles.

⟨Third test 108a⟩ ≡

```
vector<string> adjacent;

g.adjacentNodesOut("New York", back_inserter(adjacent));

cout << "List of nodes to which edges lead out of"
     << " New York: " << endl;
copy (adjacent.begin(),adjacent.end(),
      ostream_iterator<string>(cout, "   "));
cout << endl << endl;

adjacent.clear();
g.adjacentNodesIn("Los Angeles", back_inserter(adjacent));

cout << "List of nodes from which edges lead into"
     << " Los Angeles: " << endl;
copy (adjacent.begin(),adjacent.end(),
      ostream_iterator<string>(cout, "   "));
cout << endl << endl;

if ( g.isConnected("Ridgefield") )
  cout << "Ridgefield is connected." << endl;
else
  cout << "Ridgefield is not connected." << endl;

 g.addEdge("Los Angeles", "Ridgefield", 3030);

if ( g.isConnected("Ridgefield") )
  cout << "Ridgefield is connected." << endl;
else
  cout << "Ridgefield is not connected." << endl;
```

Used in parts 106a, 122b.

Next, we demonstrate the `addWeights()` function by populating a container with a list of nodes and passing an iterator to this container to the method. The method accumulates the total weight between all three cities and this is subsequently output to the screen.

⟨Fourth test 108b⟩ ≡

```
vector<string> cities(3);
cities[0] = "New York";
cities[1] = "Los Angeles";
cities[2] = "Ridgefield";

cout << "The distance between NY, LA and Ridgefield is: " <<
  g.addWeights( cities.begin(), cities.end() ) << endl;
```

Used in parts 106a, 122b.

We next demonstrate the removal of orphan nodes. To do this, we create an unconnected node named New City; we see that it is removed after the call to `removeOrphanNodes()`.

⟨Fifth test 109a⟩ ≡
```
    g.addNode("New City");

    cout << "The graph before removing orphans: "
         << endl << g << endl;
    g.removeOrphanNodes();
    cout << "The graph after removing orphans: " << endl << g;

  return 0;
 }
```

Used in parts 106a, 122b.

11.8 The AdjacencyGraphBase Class

We have also implemented another type of graph class—the adjacency-list-based graph. This is another example of the sort of graph class that may be created that conforms to `GraphInterface`. It has also, like `matrix_graph`, been designed for compliance with the STL `Container` concept.

"adjacencygraphbase.h" 109b ≡
```
#ifndef ADJACENCY_GRAPH_BASE_H
#define ADJACENCY_GRAPH_BASE_H
⟨AdjacencyGraphBase directives and header files 109c⟩
⟨This implementation's Edge class 110⟩
⟨AdjacencyGraphBase class definition and private data members 111a⟩
⟨AdjacencyGraphBase constructors and destructors 111b⟩
⟨Container methods for AdjacencyGraphBase 111c⟩
⟨AdjacencyGraphBase interface methods 112⟩
⟨Protected methods for AdjacencyGraphBase 119c⟩
⟨External overloaded output operations for AdjacencyGraphBase 120a⟩
#endif
```

This implementation, as required, derives from `GraphInterface`, the source for which must be included below. In creating an adjacency list, we will make use of the STL `list` class.

⟨AdjacencyGraphBase directives and header files 109c⟩ ≡
```
#include "graph_interface.h"
#include<list>
```

Used in part 109b.

As in `MatrixGraph`, we define an Edge class that will represent an edge joining two nodes in our graph. Since this is an adjacency-list representation, such a representation must include an identifier for the pointed-to node. The index of the pointed-to node cannot be inferred from a position in the data structure, as is the case with a matrix representation.

This additional information is stored in the private member `itsIndex`, which can be accessed via the public method `index`. Additional methods, `incIndex` and `decIndex`, are provided for decrementing and incrementing indices. Aside from creation via a copy constructor, these two methods are the only ways `itsIndex` can be modified. The reason they are needed in this graph implementation will, hopefully, become apparent when we examine `adjacency_graph`'s `removeNode` and `addNode` methods.

⟨This implementation's `Edge` class 110⟩ ≡

```
template< class Scalar >
class aEdge {
private:
  bool hidden;
  Scalar weight, itsIndex;
public:
  aEdge( Scalar index ) :
    hidden(false), weight(0), itsIndex( index ) {}
  aEdge( Scalar index, Scalar scale ) :
    hidden(false), weight(scale), itsIndex( index ) {}
  Scalar index() const { return itsIndex; }
  Scalar getWeight() const {  return weight; }
  void setWeight(Scalar wgt) { weight=wgt; }
  void decIndex() { if( itsIndex > 0 ) --itsIndex; }
  void incIndex() { ++itsIndex; }
  bool isHidden() const { return hidden; }
  bool hide() {
    if (isHidden()) return false;
    else hidden=true;
    return true;
  }

  bool unhide() {
    if (!isHidden()) return false;
    else hidden=false;
    return true;
  }
};
```

Used in part 109b.

From the below typedefs, we see that this graph representation stores edges in a vector of lists. An instance, `adjList`, of this type (`AdjType`) is allocated and remains private to this class.

⟨AdjacencyGraphBase class definition and private data members 111a⟩ ≡

```
template<class KeyType, class DataType>
class AdjacencyGraphBase :
public GraphInterface<KeyType,DataType> {
private:
  typedef KeyType Scalar;
  typedef aEdge<Scalar> EdgeType;
  typedef list<EdgeType> SublistType;
  typedef vector<SublistType> AdjType;

  AdjType          adjList;
```

Used in part 109b.

As in MatrixGraphBase, a set of constructors and destructors analogous to those given for the base classes are provided

⟨AdjacencyGraphBase constructors and destructors 111b⟩ ≡

```
public:
  explicit AdjacencyGraphBase() :
    GraphInterface() {}

  explicit AdjacencyGraphBase(bool isDi) :
    GraphInterface(isDi) {  }

  AdjacencyGraphBase ( const AdjacencyGraphBase& g ) :
    GraphInterface(g), adjList(b.adjList) {}

  ~AdjacencyGraphBase() {  }
```

Used in part 109b.

A few more methods are provided to assure compliance with the STL Container concept:

⟨Container methods for AdjacencyGraphBase 111c⟩ ≡

```
  AdjacencyGraphBase& operator= ( const AdjacencyGraphBase& g ) {
    if ( &g == this )
      return *this;
    Graph::operator=(g);
    adjList = g.adjList;
    return *this;
  }

  virtual void swap ( AdjacencyGraphBase& g ) {
    Graph::swap(g);
    adjList.swap(g.adjList);
  }
```

Used in part 109b.

Now to the heart of the matter—the implementation of the interface methods required by `graph_interface`. Some methods' code is identical to that for the matrix implementation; refer to that section for the code.

⟨`AdjacencyGraphBase` interface methods 112⟩ ≡
  ⟨Adjacency implementation of `adjacentNodesOut` 113⟩
  ⟨Adjacency implementation of `adjacentNodesIn` 114⟩
  ⟨Adjacency implementation of `isConnected` 115a⟩
  ⟨Adjacency implementation of `findWeight` 115b⟩
  ⟨Matrix implementation of `addWeights` 97b⟩
  ⟨Matrix implementation of `removeOrphanNodes` 97c⟩
  ⟨Adjacency implementation of `addNode` 116a⟩
  ⟨Matrix implementation of `addNodes` 98b⟩
  ⟨Adjacency implementation of `removeNode` 116b⟩
  ⟨Matrix implementation of `removeNodes` 98d⟩
  ⟨Adjacency implementation of `removeAllNodes` 117a⟩
  ⟨Adjacency implementation of `addEdge` 117b⟩
  ⟨Matrix implementation of `addEdges` 99c⟩
  ⟨Adjacency implementation of `removeEdge` 118a⟩
  ⟨Adjacency implementation of `getEdges` 118b⟩
  ⟨Adjacency implementation of `numEdges` 119a⟩
  ⟨Adjacency implementation of `listAll` 119b⟩

Used in part 109b.

⟨Adjacency implementation of `adjacentNodesOut` 113⟩ ≡

```
template  <class OutputIterator>
bool adjacentNodesOut(const DataType& nodeName,
                      OutputIterator o) const   {
   if( numEdges() == (Scalar)(0) )
      return false;

  if( !nodeExists(nodeName) )
    return false;

  Scalar index = getIndexOf(nodeName);

  const SublistType& sublist = adjList[index];

  if( sublist.empty() )
    return false;

  // you need to extract the index from the node
  // in this representation !!!!!!
  SublistType::const_iterator sublistIterator;
  for( sublistIterator = sublist.begin();
       sublistIterator != sublist.end(); ++sublistIterator )
      *o = nodeNameByIndex( sublistIterator->index() );

  return true;
}
```

Used in part 112.

⟨Adjacency implementation of `adjacentNodesIn` 114⟩ ≡

```
template  <class OutputIterator>
bool adjacentNodesIn(const DataType& nodeName,
                     OutputIterator o) const   {
  if( !nodeExists(nodeName) )
    return false;
  bool has_any(false);
  Scalar index = getIndexOf(nodeName);

  NodeContainer::const_iterator nodeIterator =
    nodeContainer.begin();
  SublistType::const_iterator sublistIterator;

  for ( Scalar x(0); x < adjList.size(); ++x, ++nodeIterator)
    for( sublistIterator = adjList[x].begin();
         sublistIterator != adjList[x].end();
         ++sublistIterator )
      if( x != index )  // pass over already-examined sublist
        if( (*sublistIterator).index() == index )
          {
            *o = (*nodeIterator).second;
            has_any = true;
            break;    // not a multigraph
          }
  return has_any;
}
```

Used in part 112.

⟨Adjacency implementation of `isConnected` 115a⟩ ≡

```
virtual bool isConnected(const DataType& nodeName) const  {
  if( !nodeExists(nodeName) )
    return false;   // redundant but doesn't hurt
  Scalar index = getIndexOf(nodeName);

  // ARE THERE EDGES COMING IN?
  SublistType::const_iterator sublistIterator;

  for ( Scalar x(0); x < adjList.size(); ++x )
    for( sublistIterator = adjList[x].begin();
         sublistIterator != adjList[x].end();
         ++sublistIterator )
      if( x != index )  // pass over already-examined sublist
        if( (*sublistIterator).index() == index )
          return true;

  // ARE THERE EDGES GOING OUT?
  if( !adjList[index].empty() )
    return true;

  return false;
}
```

Used in part 112.

⟨Adjacency implementation of `findWeight` 115b⟩ ≡

```
virtual Scalar findWeight( const DataType& nodeName1,
                           const DataType& nodeName2 ) const  {
  if (!nodeExists(nodeName1) | !nodeExists(nodeName2))
    return Scalar(0);

  int index1 = getIndexOf(nodeName1);
  int index2 = getIndexOf(nodeName2);

  SublistType::const_iterator i;
  for( i = adjList[index1].begin();
       i != adjList[index1].end(); ++i )
    if( (*i).index() == index2 )
      return (*i).getWeight();

  return Scalar(0); // safety valve
}
```

Used in part 112.

⟨Adjacency implementation of `addNode` 116a⟩ ≡

```
virtual void addNode ( const DataType& nodeName )   {
  Graph::addNode(nodeName);   // adds to nodeContainer
  int index = getIndexOf( nodeName );
  adjList.insert( adjList.begin() + getIndexOf( nodeName ),
                  SublistType() );

  for( AdjType::iterator i = adjList.begin();
       i < adjList.end();
       ++i )
    for( SublistType::iterator j = (*i).begin();
         j != (*i).end();
         ++j )
    {
      if( (*j).index() >= index )
        (*j).incIndex();
    }
}
```

Used in part 112.

⟨Adjacency implementation of `removeNode` 116b⟩ ≡

```
virtual void removeNode ( const DataType& nodeName )  {
  if( !nodeExists( nodeName ) )
    return;

  int index = getIndexOf( nodeName );

  for( int h = 0; h < adjList.size(); ++h )
    removeEdge( nodeNameByIndex(h), nodeName );

  Graph::removeNode(nodeName);
  adjList.erase( adjList.begin() + index );

  for( AdjType::iterator i = adjList.begin();
       i < adjList.end();
       ++i )
    for( SublistType::iterator j = (*i).begin();
         j != (*i).end();
         ++j )
    {
      if( (*j).index() > index )
        (*j).decIndex();
    }
}
```

Used in part 112.

⟨Adjacency implementation of `removeAllNodes` 117a⟩ ≡

```
virtual void removeAllNodes()  {
  Graph::removeAllNodes();
  adjList.clear();
}
```

Used in part 112.

⟨Adjacency implementation of `addEdge` 117b⟩ ≡

```
virtual bool addEdge(const DataType& nodeName1,
                     const DataType& nodeName2,
                     const Scalar weight=0)
{
  if (!nodeExists(nodeName1) || !nodeExists(nodeName2))
    return false;

  // Make sure the adjacency list has been allocated
  // (if this is the first call to addEdge)
  if (adjList.empty())
    allocateGraph();

  Scalar index1 = getIndexOf(nodeName1);
  Scalar index2 = getIndexOf(nodeName2);

  adjList[index1].push_back( EdgeType( index2, weight ) );

  return true;
}
```

Used in part 112.

⟨Adjacency implementation of `removeEdge` 118a⟩ ≡

```
virtual bool removeEdge(const DataType& nodeName1,
                        const DataType nodeName2)  {
  if (!nodeExists(nodeName1) || !nodeExists(nodeName2))
    return false;

  Scalar index1 = getIndexOf(nodeName1);
  Scalar index2 = getIndexOf(nodeName2);

  for( SublistType::iterator i = adjList[index1].begin();
       i != adjList[index1].end(); ++i )
    if( (*i).index() == index2 ) // found it! Assume no repeats
      {
        adjList[index1].erase(i);
        return true;
      }
  return false;   // not found in sublist
}
```

Used in part 112.

⟨Adjacency implementation of `getEdges` 118b⟩ ≡

```
template <class OutputIterator>
  void getEdges ( OutputIterator o ) const {
  SublistType::const_iterator sublistIterator;
  for( Scalar x(0); x < adjList.size(); ++x)
    for( sublistIterator = adjList[x].begin();
         sublistIterator != adjList[x].end();
         ++sublistIterator )
      {
        pair<DataType, DataType> p(nodeNameByIndex(x),
            nodeNameByIndex( (*sublistIterator).index() ) );

        *o = p;
     }
}
```

Used in part 112.

⟨Adjacency implementation of `numEdges` 119a⟩ ≡

```
virtual Scalar numEdges() const  {
  Scalar count=0;

  SublistType::const_iterator sublistIterator;
  for( Scalar x(0); x < adjList.size(); ++x)
    for( sublistIterator = adjList[x].begin();
         sublistIterator != adjList[x].end();
         ++sublistIterator )
      {
        ++count;
      }

  return count;
}
```

Used in part 112.

⟨Adjacency implementation of `listAll` 119b⟩ ≡

```
virtual void listAll(ostream& o) const {

  Graph::listAll(o);   // takes care of node listing

  o << endl << "Edge listings: ";
  if( adjList.size() ) o << endl;
  else o << "None." << endl;

  vector<pair<DataType, DataType> > edges;
  getEdges( back_inserter(edges) );
  for( vector<pair<DataType, DataType> >::const_iterator i =
    edges.begin();
    i < edges.end(); ++i )
      cout << (*i).first << " to " << (*i).second << endl;
}
```

Used in part 112.

A single graph allocation method is provided in the `protected` section:

⟨Protected methods for `AdjacencyGraphBase` 119c⟩ ≡

```
    void allocateGraph()  {
      if (nodeContainer.size()>0)
        for (Scalar x(0); x < nodeContainer.size(); ++x)
          adjList.push_back( SublistType() );
    }

  };  // END CLASS AdjacencyGraphBase
```

Used in part 109b.

Finally, and output operator is provided:

⟨External overloaded output operations for `AdjacencyGraphBase` 120a⟩ ≡

```
template<class KeyType, class DataType>
ostream& operator<< (ostream& o,
                     const AdjacencyGraphBase<KeyType,DataType>& g)
{
  g.listAll(o);
  return o;
}
```

Used in part 109b.

The instantiable class `adjacency_graph` parallels the `matrix_graph` class, providing functionality for compliance with the `Container` concept:

`"adjacencygraph.h"` 120b ≡

```
#ifndef ADJACENCY_GRAPH_H
#define ADJACENCY_GRAPH_H

⟨adjacency_graph directives and header files 120c⟩
⟨adjacency_graph class definition and type-definitions 121a⟩
⟨Constructors for adjacency_graph 121b⟩
⟨adjacency_graph container wrappers for STL compliance 122a⟩
#endif
```

We begin with the usual directives and file-inclusion:

⟨`adjacency_graph` directives and header files 120c⟩ ≡

```
#include "adjacencygraphbase.h"
```

Used in part 120b.

Finally, to the instantiable class for this implementation: `adjacency_graph`. Again, the class definition is very similar to that of `matrix_graph`, introducing members whose presence is required for compliance with the STL `Container` concept:

⟨`adjacency_graph` class definition and type-definitions 121a⟩ ≡

```
template <class KeyType, class DataType>
class adjacency_graph : public AdjacencyGraphBase<KeyType,DataType>
{
public:
  typedef DataType           value_type;
  typedef DataType&          reference_type;
  typedef const DataType&    const_reference;
  typedef DataType*          pointer;
  typedef ptrdiff_t          difference_type;
  typedef size_t             size_type;
  typedef graph_iterator
    <NodeContainer::iterator,DataType> iterator;
  typedef graph_iterator
    <NodeContainer::const_iterator,DataType> const_iterator;
  typedef graph_iterator
    <NodeContainer::reverse_iterator,DataType> reverse_iterator;
  typedef graph_iterator
    <NodeContainer::const_reverse_iterator,DataType>
                                           const_reverse_iterator;
```

Used in part 120b.

The constructors:

⟨Constructors for `adjacency_graph` 121b⟩ ≡

```
  explicit adjacency_graph( bool isDirected ) :
                        AdjacencyGraphBase(isDirected)
                        {  }

  explicit adjacency_graph() :
    AdjacencyGraphBase() {}

  adjacency_graph ( const adjacency_graph& g ) :
                        AdjacencyGraphBase(g) { }
```

Used in part 120b.

As with `matrix_graph`, the following members are provided for compliance with the STL `Container` concept:

⟨`adjacency_graph` container wrappers for STL compliance 122a⟩ ≡

```
    iterator begin() { return nodeContainer.begin(); }
    const_iterator begin() const
      { return nodeContainer.begin(); }
    iterator end() { return nodeContainer.end(); }
    const_iterator end() const { return nodeContainer.end(); }
    reverse_iterator rbegin() { return nodeContainer.rbegin(); }
    const_reverse_iterator rbegin() const
      { return nodeContainer.rbegin(); }
    reverse_iterator rend() { return nodeContainer.rend(); }
    const_reverse_iterator rend() const
      { return nodeContainer.rend(); }
    bool empty() const { return nodeContainer.empty(); }
    void clear() { removeAllNodes(); deAllocateGraph(); return; }
    void swap ( adjacency_graph& g ) { GraphInterface::swap(g); }
    Scalar max_size() const { return nodeContainer.max_size(); }

    adjacency_graph& operator=( const adjacency_graph& g )
    {
      AdjacencyGraphBase::operator=(g);
      return *this;
    }
  };
```

Used in part 120b.

11.9 Adjacency Graph Test Application

A program that demonstrates the functionality of `adjacency_graph` follows. The tests are identical to those performed for `matrix_graph`.

`"main2.cc"` 122b ≡

⟨Header files for the main2 program 122c⟩
⟨Declare an `adjacency_graph` object 123a⟩
⟨First test 107a⟩
⟨Second test 107b⟩
⟨Third test 108a⟩
⟨Fourth test 108b⟩
⟨Fifth test 109a⟩

First, we include the header file for the `adjacency_graph` class:

⟨Header files for the main2 program 122c⟩ ≡

```
  #include "adjacencygraph.h"
```

Used in part 122b.

Then, we declare the graph object we will use.

⟨Declare an `adjacency_graph` object 123a⟩ ≡

```
int main()
{
  adjacency_graph<unsigned int,string> g(true);
```

Used in part 122b.

See the code given for the `matrix_graph` tests.

12 Appendix B: Algorithms

Here, we present the source code for the three algorithms we have implemented in our distributed graph system.

The algorithms are :

1. `has_cycle`—tests whether a directed graph has a cycle.
2. `topo_sort`—topologically sorts the nodes of a directed graph.
3. `mst_prim`—finds the minimum spanning tree of a given graph.

"algorithms.h" 123b ≡

```
#ifndef ALGORITHMS_H
#define ALGORITHMS_H

⟨has_cycle 123c⟩
⟨topo_sort 126b⟩
⟨mst_prim 129a⟩

#endif
```

All of the below code was adapted for STL from pseudocode given in [Cormen90].

12.1 Testing for a cycle

Via a breadth-first-search of the graph, `has_cycle` determines whether or not a directed graph has a cycle.

⟨has_cycle 123c⟩ ≡

```
  ⟨hc includes and directives 124a⟩
  ⟨hc class definition and private members 124b⟩

public:
  ⟨hc constructor 124c⟩
  ⟨hc run initialization 125⟩
  ⟨hc run main loop 126a⟩
}; // end class has_cycle
```

Used in part 123b.

The algorithm requires a queue; an STL `queue` is used. The algorithm also requires node coloring. Three arbitrary constants are defined to represent these "colors."

⟨hc includes and directives 124a⟩ ≡
```
#include "adjacencygraph.h"
#include "matrixgraph.h"
#include<queue>

#define WHITE 0
#define GRAY 1
#define BLACK 2

using namespace disgraph;
```

Used in part 123c.

A pointer to a graph is maintained as a private member. This pointer will be assigned to the graph object on which the algorithm will operate.

⟨hc class definition and private members 124b⟩ ≡
```
template<class KeyType, class DataType>
class has_cycle {
private:

#ifdef MAT
typedef matrix_graph<NodeType, NodeType> GraphType;
#else
typedef adjacency_graph<NodeType, NodeType> GraphType;
#endif

  GraphType* G;
```

Used in part 123c.

⟨hc constructor 124c⟩ ≡
```

  has_cycle() {}
```

Used in part 123c.

All nodes are initially colored white (unvisited). The node at which the breadth-first search begins (by default the node which has been assigned an index of 0 by the graph object) is then colored white (visited).

⟨hc run initialization 125⟩ ≡

```
bool run( GraphType& g ) {
  G = &g;

  int num_vertices = G->size();
  if( !num_vertices ) return false;

  int* color = new int[num_vertices];

  for( int i = 0; i < num_vertices; ++i )
    color[i] = WHITE;
  queue<DataType> Q;

  color[0] = GRAY;
  Q.push( 0 );
```

Used in part 123c.

The breadth-first search proceeds. Nodes that have been examined are colored gray, and nodes that are "finished"—those that will no longer participate in the search— are colored black. If a black node is found during the search, the graph has a cycle; `true` is immediately returned.

If the search has examined all nodes without encountering a node that it has previously colored black, no cycles exist in the graph—`false` is therefore returned.

⟨hc run main loop 126a⟩ ≡
```
    DataType u, v;
    while( !Q.empty() ) {
      u = Q.front();

      vector<DataType> neighbors;
      G->adjacentNodesOut( u, back_inserter(neighbors) );

      for( vector<DataType>::iterator p = neighbors.begin();
           p < neighbors.end(); ++p )
        {
          v = *p;
          if( color[G->getIndexOf(v)] == BLACK || v == u ) {
            delete color;
          return true;
        }
       if( color[G->getIndexOf(v)] == WHITE ) {
              color[G->getIndexOf(v)] = GRAY;
              Q.push(v);
          }
      }
      Q.pop();
      color[G->getIndexOf(u)] = BLACK;
    }
    delete color;
    return false;
  } // end run()
```

Used in part 123c.

12.2 Topological Sort

This implementation of the standard topological sort algorithm [Cormen90] uses depth-first search discovery times to determine the ordering of nodes to be returned. If the graph has a cycle, the ordering of returned nodes is simply an ordering of their depth-first search discovery times.

⟨topo_sort 126b⟩ ≡
```
  ⟨ts includes and directives 127a⟩
  ⟨ts private data members 127b⟩
  ⟨void DFS_Visit 127c⟩
public:
  ⟨ts constructor 128a⟩
  ⟨ts run 128b⟩
}; // end class topo_sort
```

Used in part 123b.

This implementation requires only two "colors."

⟨ts includes and directives 127a⟩ ≡

```
#include "adjacencygraph.h"
#include "matrixgraph.h"

#define WHITE 0
#define GRAY 1
```

Used in part 126b.

A pointer `color` will be assigned to an array holding the current colorings of the nodes. As they are discovered, nodes are stored in `returnNodes`.

⟨ts private data members 127b⟩ ≡

```
template<class KeyType, class DataType>
class topo_sort {
private:
#ifdef MAT
typedef matrix_graph<NodeType, NodeType> GraphType;
#else
typedef adjacency_graph<NodeType, NodeType> GraphType;
#endif
  GraphType* G;

  int* color;
  vector<DataType> return_nodes;
```

Used in part 126b.

`DFS_Visit` recursively visits the nodes of the graph and stores discovered nodes in `returnNodes`.

⟨`void DFS_Visit` 127c⟩ ≡

```
  void DFS_Visit( int u ) {
    color[G->getIndexOf(u)] = GRAY;

    vector<DataType> neighbors;

    G->adjacentNodesOut( u, back_inserter( neighbors ) );
    for( vector<DataType>::iterator i = neighbors.begin();
         i < neighbors.end(); ++i )
      if( color[G->getIndexOf(*i)] == WHITE )
        DFS_Visit( *i );

    return_nodes.push_back( u );
  }
```

Used in part 126b.

⟨ts constructor 128a⟩ ≡

```
topo_sort() {}
```

Used in part 126b.

A node at which to begin the search must be provided when running this algorithm. The body of the below method simply performs some bookkeeping, calls `DFS_Visit`, and then supplies the discovered nodes to the passed-in output iterator.

⟨ts `run` 128b⟩ ≡

```
template<class OutputIterator>
void run( GraphType& g,
      const DataType& u, OutputIterator o ) {
  // uses DFS
  // u must exist in the graph
  G = &g;

  int num_vertices = G->size();
  if( !num_vertices )
    return;

  color = new int[num_vertices];
  return_nodes.clear();

  for( int i = 0; i < num_vertices; ++i )
    color[i] = WHITE;

  DFS_Visit( u );

  delete color;

  for( int i = return_nodes.size() - 1; i >= 0; --i )
    *o = return_nodes[i];
}
```

Used in part 126b.

12.3 Finding a Minimum Spanning Tree

The below code to find the minimum spanning tree of a graph is derived from Prim's algorithm as described in [Cormen90].

```
⟨mst_prim 129a⟩ ≡
    ⟨mst includes and directives 129b⟩
    ⟨second_less predicate 130a⟩

  template<class KeyType, class DataType>
  class mst_prim {
  private:
    ⟨mst private data members 130b⟩
    ⟨mst init 131a⟩
    ⟨mst load_return_graph 131b⟩
    ⟨mst process_edge 132a⟩

  public:
    ⟨mst constructor 132b⟩
    ⟨mst run 133⟩
  }; // end class mst_prim
```

Used in part 123b.

Some representation of "infinity" must be provided. Any negative number will serve this purpose, assuming that all edge weights of the graph are nonnegative.

```
⟨mst includes and directives 129b⟩ ≡
  #include "adjacencygraph.h"
  #include "matrixgraph.h"

  #define INFINITY -1
```

Used in part 129a.

Each node must be associated with a key. Here, this is done using a `pair` in which the first element is the node and the second is its key.

The algorithm depends on the ordering of nodes based on their keys. Such an ordering is enabled by the below predicate which compares the `second` member of a pair. Here this ordering must take our representation of infinity into account; any key with a value of `INFINITY` is determined to be greater than any other key.

⟨second_less predicate 130a⟩ ≡

```
template<class Pair>
class second_less
{
public:
  typedef Pair first_argument_type;
  typedef Pair second_argument_type;
  typedef bool result_type;

  bool operator() (const Pair& p1, const Pair& p2) const {
    if( p2.second == INFINITY )
      return true;
    if( p1.second == INFINITY )
      return false;
    return (p1.second < p2.second);
  }
};
```

Used in part 129a.

Several private data structures are maintained for use during the algorithm's execution.

⟨mst private data members 130b⟩ ≡

```
#ifdef MAT
typedef matrix_graph<NodeType, NodeType> GraphType;
#else
typedef adjacency_graph<NodeType, NodeType> GraphType;
#endif

  GraphType* G;
  GraphType* return_graph;

  typedef pair<DataType, int> nodeWithKey;
  typedef pair<DataType, DataType> anEdge;
  typedef vector<nodeWithKey> HeapType;
  HeapType Q;

  vector<anEdge> returnEdges;
  vector<DataType> returnNodes;
```

Used in part 129a.

After ensuring that all private data structures have been cleared, `init` assigns the key `INFINITY` to all nodes except for the root node.

⟨mst `init` 131a⟩ ≡

```
bool init( const int& r ) {
  return_graph = new GraphType( true );
  if( !(G->size()) || !(G->numEdges()) )
    return false;

  Q.erase( Q.begin(), Q.end() );
  returnNodes.erase( returnNodes.begin(), returnNodes.end() );
  returnEdges.erase( returnEdges.begin(), returnEdges.end() );

  GraphType::iterator i = G->begin();
  for( ; i != G->end(); ++i )
    if( *i == r )
      Q.push_back( nodeWithKey( *i, 0 ) );
    else
      Q.push_back( nodeWithKey(*i, INFINITY) );

  sort( Q.begin(), Q.end(), second_less<nodeWithKey>() );
  return true;
}
```

Used in part 129a.

The graph instance to be returned must be populated with the data gathered throughout algorithm execution:

⟨mst `load_return_graph` 131b⟩ ≡

```
void load_return_graph() {
  return_graph->addNodes( returnNodes.begin(),
                          returnNodes.end() );

  for( int k = 0; k < returnEdges.size(); ++k )
    return_graph->addEdge( returnEdges[k].first,
                           returnEdges[k].second,
                           G->findWeight( returnEdges[k].first,
                                          returnEdges[k].second ) );
}
```

Used in part 129a.

The majority of the processing done by this algorithm occurs in the below method.

⟨mst process_edge 132a⟩ ≡

```
void process_edge( const DataType& u, const DataType& v ) {
  int key_of_v;
  HeapType::iterator it = Q.begin();

  for( ; it < Q.end(); it++ )
    if( (*it).first == v )  {
      key_of_v = (*it).second;
      break;
    }

  if( it != Q.end() )  {
    int uv_weight = 0; // so will work w/ directed ...
    if( !(uv_weight = G->findWeight( u, v )) )
      uv_weight = G->findWeight( v, u );

    if( uv_weight < key_of_v || key_of_v == INFINITY )  {
      // update edge list
      for( vector<anEdge>::iterator zz = returnEdges.begin();
           zz < returnEdges.end(); ++zz )
        if( (*zz).second == v )
          returnEdges.erase( zz );

      returnEdges.push_back( pair<DataType, DataType>( u, v ) );
      (*it).second = uv_weight;
    }
  }
}
```

Used in part 129a.

⟨mst constructor 132b⟩ ≡

```
mst_prim() {}
```

Used in part 129a.

The caller must provide a root node for the minimum spanning tree.

⟨mst run 133⟩ ≡

```
// r is the root of the MST-tree
GraphType& run( GraphType& g, const DataType& r ) {
  G = &g;

  if( !init( r ) )
    return *return_graph;

  DataType u, v;
  int w;    // given weight (again, scalar assumed)

  while( Q.size() > 0 )  {
    u = (*(Q.begin())).first;
    Q.erase( Q.begin() );  // u = EXTRACT-MIN(Q)

    returnNodes.push_back( u );   // mark as already handled

    vector<DataType> neighbors;
    neighbors.erase( neighbors.begin(), neighbors.end() );
    G->adjacentNodesOut( u, back_inserter( neighbors ) );
    G->adjacentNodesIn( u, back_inserter( neighbors ) );

    for( vector<DataType>::iterator v = neighbors.begin();
         v < neighbors.end(); ++v )
      {
        process_edge( u, *v );
        sort( Q.begin(), Q.end(), second_less<nodeWithKey>() );
      }
  } // end while

  load_return_graph();
  return *return_graph;
} // end run()
```

Used in part 129a.

References

[Baker97] Sean Baker. *CORBA Distributed Objects*, Addison-Wesley, 1997.

[Cormen90] T. Cormen, C. Lieserson, R. Rivest. *Introduction to Algorithms*, MIT Press, 1990.

[Gamma95] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[GTL99] GTL : Graph Template Library. *http://www.fmi.uni-passau.de/ himsolt/Graphlet/GTL/*.

[Knuth93] D. E. Knuth. *The Stanford GraphBase : A Platform for Combinatorial Computing*, Addison Wesley, 1993.

[MICO] MICO : MICO Is CORBA. *http://www.mico.org*.

[Musser96] D. Musser, A. Saini. *STL Tutorial and Reference Guide : C++ Programming with the Standard Template Library*, Addison-Wesley, 1996.

[OMG] The Object Management Group. *http://www.omg.org*.

[Patterson96] D. Patterson, J. Hennessy. *Computer Architecture: A Quantitative Approach*, 2nd Edition, Morgan Kaufmann, 1996.

[Roemer99] K. Roemer, A. Puder. *MICO 2.2 Implementation Manual (Version 2.2.5)*, available at *http://www.mico.org*.

[Roemer99b] K. Roemer. From an email discussion.

[Stroustrup97] B. Stroustrup. *The C++ Programming Language*, 3rd Edition, Addison-Wesley, 1997.