```
#include <iostream>
#include <fstream>
#include "d_matrix.h"
#include "d_except.h"
#include <list>
#include <stack>

using namespace std;

typedef int WeightType;
typedef int NodeType;

int const MaxNumNodex = 9999;

typedef int NodeWeight;
typedef int EdgeWeight;

class node //Vertex
{
   public:
      node();
      node(const node &);
      node &operator=(const node &);

      void setId(int i);
      int getId() const;

      void setWeight(NodeWeight);
      NodeWeight getWeight() const;

      void setNode(int, NodeWeight, bool, bool);

      void mark();
      void unMark();
      bool isMarked() const;

      void visit();
      void unVisit();
      bool isVisited() const;

   private:
      int id; √      1, 2, 3, . . .
      NodeWeight weight;
      bool marked;
      bool visited;
};

node::node()
// Constructor, creates a new, uninitialized node. Id is initialized
// elsewhere (by the graph constructor).
{
   unMark();
   unVisit();
   setWeight(0);
}
```

```
node::node(const node &n)
// Copy constructor
{
   setNode(n.getId(), n.getWeight(), n.isMarked(), n.isVisited());
}

node &node::operator=(const node &n)
// Overloaded assignment operator.
{
   setNode(n.getId(), n.getWeight(), n.isMarked(), n.isVisited());
   return *this;
}

NodeWeight node::getWeight() const
// Return node's weight
{
   return weight;
}

void node::setWeight(NodeWeight w)
// Set node's weight to w.
{
   weight = w;
}

void node::setId(int i)
// Set node's id to i.  Throws an exception if i < 0.
{
   if (i < 0)
      throw rangeError("Bad value in node::setId");

   id = i;
}

int node::getId() const
// Return node's id
{
   return id;
}

void node::setNode(int id, NodeWeight w = 0, bool m = false, bool v = false)
// Initialize a node;
{
   setId(id);
   setWeight(w);

   if (m)
      mark();
   else
      unMark();

   if (v)
      visit();
   else
      unVisit();
}
```

```
void node::mark()
   // Mark node.
{
   marked = true;
}

void node::unMark()
   // Un-mark node.
{
   marked = false;
}

bool node::isMarked() const
   // Return true if node is marked, and false otherwise.
{
   return marked;
}

void node::visit()
   // Set visited to true;
{
   visited = true;
}

void node::unVisit()
   // Set visited to false;
{
   visited = false;
}

bool node::isVisited() const
   // Return true if node is visited, and false otherwise.
{
   return visited;
}


ostream &operator<<(ostream &ostr, const node &n)
{
   ostr << "node: " << n.getId() << " weight: " << n.getWeight()
        << " visited: " << n.isVisited() << " marked " << n.isMarked() << endl;

   return ostr;
}

class edge
{
   public:
                s     d           1
      edge();
   ✓  edge(int, int, EdgeWeight = 0);
      edge(const edge &);
      edge &operator=(const edge &);

      void setWeight(EdgeWeight);
      EdgeWeight getWeight() const;
      int getSource() const;
      int getDest() const;

      void setValid();
      void setInvalid();
      bool isValid() const;

      void mark();
      void unMark();
      bool isMarked() const;

      void visit();
      void unVisit();
      bool isVisited() const;

      void setEdge(int, int, EdgeWeight);

   private:
      int source;  vi id   vi → vj
      int dest;    vj id
      EdgeWeight weight;
      bool valid;         // equals true if edge is valid, otherwise the
      bool visited;
      bool marked;
      // edge is invalid
};

edge::edge()
// Constructor, sets edge to invalid, unmarked and unvisited.
{
   setInvalid();
   unMark();
   unVisit();
}

edge::edge(int i, int j, EdgeWeight w)
// Constructor creates an edge with weight w, and marks the edge as valid,
    unvisited
// and unmarked.
{
   setEdge(i,j,w);
   unMark();
   unVisit();
}

edge::edge(const edge &e)
// Copy constructor.  Also copies visited and marked state.
{
   setEdge(e.source, e.dest, e.getWeight());

   if (e.isValid())
      setValid();
   else
      setInvalid();
   if (e.isVisited())
      visit();
   else
      unVisit();

   if (e.isMarked())
      mark();
   else
      unMark();
}

edge &edge::operator=(const edge &e)
// Overloaded assignment operator
{
   setEdge(e.source, e.dest, e.getWeight());

   if (e.isValid())
      setValid();
   else
      setInvalid();

   if (e.isVisited())
      visit();
   else
      unVisit();

   if (e.isMarked())
      mark();
   else
      unMark();

   return *this;
}

void edge::setEdge(int i, int j, EdgeWeight w = 0)
// Initialize edge with source, destination and weight and mark edge
// as valid.  Do not change visited or marked state.
{
   source = i;
   dest = j;
   weight = w;
   setValid();
}

void edge::setWeight(EdgeWeight w)
// Set edge weight to w.
{
   weight = w;
}

EdgeWeight edge::getWeight() const
// Return edge weight.
{
   return weight;
}
```

```
EdgeWeight edge::getSource() const
// Return source node;
{
   return source;
}

EdgeWeight edge::getDest() const
// Return destination node.
{
   return dest;
}

void edge::setValid()
// Set the edge as valid.
{
   valid = true;
}

void edge::setInvalid()
// Set the edge as invalid.
{
   valid = false;
}

bool edge::isValid() const
// Return true if edge is valid.  Otherwise return false;
{
   return valid;
}

void edge::mark()
// Mark edge
{
   marked = true;
}

void edge::unMark()
// Un-mark edge
{
   marked = false;
}

bool edge::isMarked() const
// Return true if edge is marked, and false otherwise.
{
   return marked;
}

void edge::visit()
// Set visited to true;
{
   visited = true;
}

void edge::unVisit()
// Set visited to false;
```

```
{
   visited = false;
}

bool edge::isVisited() const
// Return true if edge is visited, and false otherwise.
{
   return visited;
}

ostream &operator<<(ostream &ostr, const edge &e)
// Print all edge information for a valid edge;
{
   cout << "edge (" << e.getSource() << "," << e.getDest() << "): ";
   cout << " weight: " << e.getWeight() << " visited: " << e.isVisited()
    << " marked " << e.isMarked() << endl;

   return ostr;
}

class graph
{
  public:
   graph();   // empty graph
   graph(int n);
   graph(ifstream &fin);
   graph(const graph &);
   graph &operator=(const graph &);

   void addEdge(int i, int j, ~~NodeWeight w =~~ 0);
   void removeEdge(int i, int j);

   int addNode(NodeWeight w = 0);   // return node id
   int addNode(node n);

   void setEdgeWeight(int i, int j, EdgeWeight w = 0);
   EdgeWeight getEdgeWeight(int i, int j) const;

   NodeWeight getTotalNodeWeight();
   EdgeWeight getTotalEdgeWeight();

   void setNodeWeight(int i, NodeWeight w = 0);
   NodeWeight getNodeWeight(int i) const;
                         s              d
   bool isEdge(NodeType i, NodeType j) const;   // check neighbors.
   int numNodes() const;                        true. v_i → v_j
   int numEdges() const;

   node &getNode(int);
   const node &getNode(int) const;
   edge &getEdge(int i,int j);
   const edge &getEdge(int i, int j) const;

   void printNodes() const;
   void printEdges() const;
```

```
      void mark(int i);
      void mark(int i, int j);
      void unMark(int i);
      void unMark(int i, int j);
      bool isMarked(int i) const;
      bool isMarked(int i, int j) const;
      void clearMark();
      bool allNodesMarked();   set node i as visited!
                                     node visited.
      void visit(int i); ✓ //
      void visit(int i, int j);
      void unVisit(int i); ✓ // set node i as unvisited
      void unVisit(int i, int j);
      bool isVisited(int i, int j) const;
      bool isVisited(int i) const; ✓ // check if visited
      void clearVisit(); ✓ ☆
      bool allNodesVisited();

    private:
      matrix<edge> edges;   ||
      vector<node> nodes;   ||
      int NumEdges;

   };

   graph::graph()
      :edges(0,0), nodes(0)
   // Constructor that creates an empty graph. graph containing n nodes and no
      edges.
   {
      NumEdges = 0;
   }

   graph::graph(int n)
   // Constructor that creates a graph containing n nodes and no edges.
   // Edges and nodes are initialized by their constructors.
   {
      for (int i = 0; i < n; i++)
         addNode();

      NumEdges = 0;
   }

   graph::graph(ifstream &fin)
   // Construct a new graph using the data in fin.
   {
      int n, i, j, w;
      fin >> n;

      // Add nodes.
      for (int i = 0; i < n; i++)
         addNode();

      NumEdges = 0;

      while (fin.peek() != '.')
      {
         fin >> i >> j >> w;
         addEdge(i, j, w);
      }
   }

   graph::graph(const graph &g)
   // Graph copy constructor.
   {
      NumEdges = 0;

      // Create a temporary node to pass to nodes::resize to initialize
      // new nodes.  This avoids the exception that is thrown by
      // node::setId which is called by the node copy constructor.  The
      // temporary node is overwritten later in this function.

      node tempNode;
      tempNode.setId(0);

      nodes.resize(g.numNodes(),tempNode);
      edges.resize(g.numNodes(),g.numNodes());

      // Copy the nodes using the overloaded assignment operator.
      for (int i = 0; i < numNodes(); i++)
         nodes[i] = g.nodes[i];

      // Copy the edges using the overloaded assignment operator.
      for (int i = 0; i < numNodes(); i++)
         for (int j = 0; j < numNodes(); j++)
        edges[i][j] = g.edges[i][j];
   }

   graph &graph::operator=(const graph &g)
   // Graph assignment operator.
   {
      // Create a temporary node to pass to nodes::resize to initialize
      // new nodes.  This avoids the exception that is thrown by
      // node::setId which is called by the node copy constructor.  The
      // temporary node is overwritten later in this function.

      node tempNode;
      tempNode.setId(0);

      nodes.resize(g.numNodes(),tempNode);
      edges.resize(g.numNodes(), g.numNodes());

      // Copy the nodes.
      for (int i = 0; i < numNodes(); i++)
         nodes[i] = g.nodes[i];

      // Copy the edges.
      for (int i = 0; i < numNodes(); i++)
         for (int j = 0; j < numNodes(); j++)
        edges[i][j] = g.edges[i][j];

      return *this;
```

```
}

int graph::addNode(NodeWeight w)
// Add a new node with weight w.  Also increase the size of the edges
// matrix.  Return the index of the new node.
{
   node n;
   n.setNode(numNodes(),w);
   nodes.push_back(n);

   edges.resize(numNodes(),numNodes());
   return numNodes()-1;
}

int graph::addNode(node n)
// Add a new node that is a copy of node n (note that the node is
// passed by value).  Also increase the size of the edges matrix.
// Return the index of the new node.
{
   nodes.push_back(n);

   edges.resize(numNodes(),numNodes());
   return numNodes()-1;
}

void graph::addEdge(int i, int j, NodeWeight w)
// Add an edge of weight w from node i to node j.  Throws an exception
// if i or j is too small or large.  Does not allow duplicate edges
// to be added.
{
   if (i < 0 || i >= numNodes() || j < 0 || j >= numNodes())
      throw rangeError("Bad value in graph::addEdge");

   if (!isEdge(i,j))
      edges[i][j] = edge(i,j,w);
   NumEdges++;
}

void graph::removeEdge(int i, int j)
// Remove the edge between node i and node j.  Throws an exception if
// i or j is too large or too small, or if the edge does not exist.
{
   if (i < 0 || i >= numNodes() || j < 0 || j >= numNodes())
      throw rangeError("Bad value in graph::removeEdge");

   if (!isEdge(i,j))
      throw rangeError("Edge does not exist in graph::removeEdge");


   edges[i][j].setInvalid();
   NumEdges--;
}

EdgeWeight graph::getEdgeWeight(int i, int j) const
// Return the weight of the edge between node i and node j. Throws an
// exception if i or j is too small or too large, or if the edge does
// not exist.
{
   if (i < 0 || i >= numNodes() || j < 0 || j >= numNodes())
      throw rangeError("Bad value in graph::getEdgeWeight");

   if (!isEdge(i,j))
      throw rangeError("Edge does not exist in graph::getEdgeWeight");

   return edges[i][j].getWeight();
}

void graph::setEdgeWeight(int i, int j, EdgeWeight w)
// Sets the weight of the arc/edge from node i to node j to w.  Throws
// an exception if ir or j is too small or too large, or if the edge
// does not exist.
{
   if (i < 0 || i >= numNodes() || j < 0 || j >= numNodes())
      throw rangeError("Bad value in graph::setEdgeWeight");

   edges[i][j].setWeight(w);
}

NodeWeight graph::getNodeWeight(int i) const
// Returns the weight of node i.  Throws an exception if i is too
// small or too large.
{
   if (i < 0 || i >= numNodes())
      throw rangeError("Bad value in graph::getNodeWeight");

   return nodes[i].getWeight();
}

void graph::setNodeWeight(int i, NodeWeight w)
// Sets the weight of node i to w.  Throws an exception if i is too
// small or too large.
{
   if (i < 0 || i >= numNodes())
      throw rangeError("Bad value in graph::setNodeWeight");

   nodes[i].setWeight(w);
}

NodeWeight graph::getTotalNodeWeight()
// Return the total node weight.
{
   NodeWeight weight = 0;

   for (int i = 0; i < numNodes(); i++)
      weight = weight + nodes[i].getWeight();

   return weight;
}

EdgeWeight graph::getTotalEdgeWeight()
// Return the total edge weight.
{
   EdgeWeight weight = 0;

   for (int i = 0; i < numNodes(); i++)
      for (int j = 0; j < numNodes(); j++)
         if (isEdge(i,j))
        weight = weight + edges[i][j].getWeight();

   return weight;
}

bool graph::isEdge(NodeType i, NodeType j) const
// Return true if there is an arc/edge from node i to node j, and
// false otherwise.  Throws an exception if i is too small or too
// large.
{
   if (i < 0 || i >= numNodes() || j < 0 || j >= numNodes())
      throw rangeError("Bad value in graph::isEdge");

   return edges[i][j].isValid();
}

int graph::numNodes() const
// Return the number of nodes.
{
   return nodes.size();
}

int graph::numEdges() const
// Return the number of edges.
{
   return NumEdges;
}

void graph::printNodes() const
// Print all nodes.
{
   cout << "Num nodes: " << numNodes() << endl;

   for (int i = 0; i < numNodes(); i++)
      cout << getNode(i);
}

void graph::printEdges() const
// Print edge information about the graph.
{
   cout << "Num edges: " << numEdges() << endl;

   for (int i = 0; i < numNodes(); i++)
      for (int j = 0; j < numNodes(); j++)
      {
     if (edges[i][j].isValid())
        cout << getEdge(i,j);
      }
}

ostream &operator<<(ostream &ostr, const graph &g)
```

```
// Print all information about the graph.
{
   cout << "------------------------------------------------" << endl;
   g.printNodes();
   cout << endl;
   g.printEdges();
   cout << endl;

   return ostr;
}

node &graph::getNode(int i)
// Return a reference to the ith node.  Throws an exception if i is
// too small or too large.
{
   if (i < 0 || i >= numNodes())
      throw rangeError("Bad value in graph::getNode");

   return nodes[i];
}

const node &graph::getNode(int i) const
// Return a reference to the ith node.  Throws an exception if i is
// too small or too large.
{
   if (i < 0 || i >= numNodes())
      throw rangeError("Bad value in graph::getNode");

   return nodes[i];
}

edge &graph::getEdge(int i, int j)
// Return a reference to the edge connecting nodes i and j.  If i is
// too small or too large, or if the edge does not exist, throws an
// exception.
{
   if (i < 0 || i >= numNodes() || j < 0 || j >= numNodes())
      throw rangeError("Bad value in graph::getEdge");

   if (!isEdge(i,j))
      throw rangeError("Edge does not exist in graph::getEdge");

   return edges[i][j];
}

const edge &graph::getEdge(int i, int j) const
// Return a reference to the edge connecting nodes i and j.  If i is
// too small or too large, or if the edge does not exist, throws an
// exception.
{
   if (i < 0 || i >= numNodes() || j < 0 || j >= numNodes())
      throw rangeError("Bad value in graph::getEdge");

   if (!isEdge(i,j))
      throw rangeError("Edge does not exist in graph::getEdge");
   return edges[i][j];
}

void graph::mark(int i)
// Mark node i.  Throws an exception if i is too large or too small.
{
   if (i < 0 || i >= numNodes())
      throw rangeError("Bad value in graph::getEdge");

   nodes[i].mark();
}

void graph::mark(int i, int j)
// Mark edge (i,j).  Throws an exception if (i,j) is not an edge.
{
   if (!isEdge(i,j))
      throw rangeError("Bad value in graph::mark");

   edges[i][j].mark();
}

void graph::unMark(int i)
// unMark node i.  Throws an exception if i is too large or too small.
{
   if (i < 0 || i >= numNodes())
      throw rangeError("Bad value in graph::getEdge");

   nodes[i].unMark();
}

void graph::unMark(int i, int j)
// unMark edge (i,j).  Throws an exception if (i,j) is not an edge.
{
   if (!isEdge(i,j))
      throw rangeError("Bad value in graph::unMark");

   edges[i][j].unMark();
}

bool graph::isMarked(int i) const
// Return true if node i is marked.  Otherwise return false.  Throws an
// exception if i is too large or too small.
{
   if (i < 0 || i >= numNodes())
      throw rangeError("Bad value in graph::getEdge");

   return nodes[i].isMarked();
}

bool graph::isMarked(int i, int j) const
// Return true if edge (i,j) node is marked.  Otherwise return false.
// Throws an exception if (i,j) is not an edge.
{
   if (!isEdge(i,j))
      throw rangeError("Bad value in graph::isMarked");
   return edges[i][j].isMarked();
}

void graph::clearMark()
// Set all nodes and edges as unmarked.
{
   for (int i = 0; i < numNodes(); i++)
      nodes[i].unMark();

   for (int i = 0; i < numNodes(); i++)
      for (int j = 0; j < numNodes(); j++)
     if (isEdge(i,j))
        edges[i][j].unMark();
}

void graph::visit(int i)
// Mark node i as visited.  Throws an exception if i is too large or
// too small.
{
   if (i < 0 || i >= numNodes())
      throw rangeError("Bad value in graph::getEdge");

   nodes[i].visit();
}

void graph::visit(int i, int j)
// Mark edge (i,j) as visited.  Throws an exception (i,j) is not an
// edge.
{
   if (!isEdge(i,j))
      throw rangeError("Bad value in graph::visite");

   edges[i][j].visit();
}

void graph::unVisit(int i)
// Set node i as unvisited.  Throws an exception if i is too large or
// too small.
{
   if (i < 0 || i >= numNodes())
      throw rangeError("Bad value in graph::getEdge");

   nodes[i].unVisit();
}

void graph::unVisit(int i, int j)
// Set edge (i,j) as unvisited.  Throws an exception if (i,j) is not
// an edge.
{
   if (!isEdge(i,j))
      throw rangeError("Bad value in graph::unVisit");

   edges[i][j].unVisit();
}

bool graph::isVisited(int i) const
```

```
// Return true if node has been visited.  Otherwise return false.  Throws an
// exception if i is too large or too small.
{
   if (i < 0 || i >= numNodes())
      throw rangeError("Bad value in graph::getEdge");

   return nodes[i].isVisited();
}

bool graph::isVisited(int i, int j) const
// Return true if edge (i,j) has been visited.  Otherwise return
// false.  Throws an exception if (i,j) is not an edge.
{
   if (!isEdge(i,j))
      throw rangeError("Bad value in graph::isVisited");

   return edges[i][j].isVisited();
}

void graph::clearVisit()
// Set all nodes and edges as unvisited.
{
   for (int i = 0; i < numNodes(); i++)
      nodes[i].unVisit();

   for (int i = 0; i < numNodes(); i++)
      for (int j = 0; j < numNodes(); j++)
     if (isEdge(i,j))
        unVisit(i,j);
}

bool graph::allNodesVisited()
// Return true if all nodes have been visited.  Otherwise return
// false.
{
   for (int i = 0; i < numNodes(); i++)
      if (!isVisited(i))
     return false;

   return true;
}

bool graph::allNodesMarked()
// Return true if all nodes have been marked.  Otherwise return
// false.
{
   for (int i = 0; i < numNodes(); i++)
      if (!isMarked(i))
     return false;

   return true;
}
```