CPS222 - DATA STRUCTURES AND ALGORITHMS
EXAMPLES OF CODE FOR VARIOUS GRAPH ALGORITHMS

Note: Two different representations are developed - adjacency matrix (class GraphM) and adjacency list (class GraphL). Some algorithms are done for the former and some for the latter - in most cases not because of any inherent benefit of one representation versus the other for that algorithm, but only to provide demonstrations of both representations. (The exception is Warshall's algorithm, which is a lot simpler on the matrix representation as presented below.)

```
/* Demonstration of some graph algorithms.  Selected algorithms are presented for two
 * representations for graphs: adjacency matrix and adjacency list.
 * Copyright (c) 2001, 2003 - Russell C. Bjork
 */

#include <string>
#include <iostream>
#include <fstream>
#include <iomanip>
#include <map>
#include <list>
#include <stack>
#include <queue>
using namespace std;

/*****************************************************************************
 * The following class represents a graph by using an adjacency matrix
 *****************************************************************************/

class GraphM
{
    public:
        // Constructor - construct a representation from a text file
        GraphM(istream & source);
        // DFS traversal from a specified starting vertex -
        // write vertex names to a text file
        void dfs(int start, ostream & output) const;
        // Convert this graph into its transitive closure
        void closure();
        // Destructor
        ~GraphM()
        {
            delete [] _vertexName;
            for (int i = 0; i < _numberOfVertices; i ++)
                delete [] _edge[i];
            delete [] _edge;
        }
    private:
        int _numberOfVertices;
        string * _vertexName;
        bool * * _edge;
        // **** IF REPRESENTING A WEIGHTED GRAPH (NETWORK), MAKE THE TYPE ****
        // **** OF THE ABOVE A NUMERIC TYPE (E.G. double) INSTEAD OF bool ****
};
```

```
/*****************************************************************************
 * The following class represents a graph by using an adjacency list
 *****************************************************************************/

class GraphL
{
    public:

        // Constructor - construct a representation from a text file
        GraphL(istream & source);

        // BFS traversal from a specified starting vertex
        // - write vertex names to a text file
        void bfs(int start, ostream & output) const;

        // Write out the vertices in topological order
        void topsort(ostream & output) const;

        // Destructor
        ~GraphL()
        { delete [] _vertex; }

    private:

        /* This inner class represents one edge of the graph */
        class Edge
        {
            public:

                Edge(int head)
                : _head(head)
                { }

                int _head;
                // **** IF REPRESENTING A WEIGHTED GRAPH (NETWORK) ADD THE ****
                // **** FOLLOWING, USING SOME NUMERIC TYPE (E.G.double) ****
                // **** ALSO ADD A weight PARAMETER TO THE CONSTRUCTOR ****
                // double _weight;
        };

        /* This inner class represents one vertex of the graph */
        class Vertex
        {
            public:

                string _name;
                typedef list < Edge > EdgeList;
                EdgeList _edges;
        };

        int _numberOfVertices;
        Vertex * _vertex;
};
```

```
/*****************************************************************************
 * Read a graph from a file into an adjacency matrix
 * File format: first line: number of vertices (N), number of edges (E)
 *              next N lines: names of N vertices
 *              next E lines: edges - each specified by tail vertex name,
 *                            head vertex name.
 * This file format and code can be used for either an undirected graph or a
 * digraph - see line marked with ** below.  It can also be used for a network
 * if we add a weight to each edge line - see lines marked with **** below
 *****************************************************************************/

GraphM::GraphM(istream & source)
{
    // Read number of vertices and number of edges
    int numberOfEdges;
    source >> _numberOfVertices >> numberOfEdges;
    // Read in names of vertices.  Save each vertex name permanently
    // in the _vertexName array, and also temporarily in a map to
    // facilitate reading in the edge descriptions by vertex name
    _vertexName = new string[_numberOfVertices];
    map < string, int > nameMap;
    for (int i = 0; i < _numberOfVertices; i ++)
    {
        string name;
        source >> name;
        _vertexName[i] = name;
        nameMap[name] = i;
    }
    // Read in edges.  Save each edge in adjacency matrix
    _edge = new bool * [ _numberOfVertices ];
    for (int i = 0; i < _numberOfVertices; i ++)
        _edge[i] = new bool [_numberOfVertices];
    // **** CHANGE bool TO NUMERIC TYPE (E.G. double) FOR A NETWORK ****

    for (int i = 0; i < _numberOfVertices; i ++)
        for (int j = 0; j < _numberOfVertices; j ++)
            _edge [i] [j] = false;
    // **** REPLACE false IN ABOVE LINE BY "infinity" FOR A NETWORK ****

    for (int i = 0; i < numberOfEdges; i ++)
    {
        string tail, head;
        source >> tail >> head;
        int tailIndex = nameMap[tail];
        int headIndex = nameMap[head];
        // **** FOR A NETWORK, ADD THE FOLLOWING LINES, AND CHANGE true ****
        // **** IN THE SUBSEQUENT LINE(S) TO weight
        // double weight;
        // source >> weight;
        _edge [ tailIndex ] [ headIndex ] = true;
        // ** THE FOLLOWING LINE WOULD BE OMITTED FOR A DIGRAPH **
        _edge [ headIndex ] [ tailIndex ] = true;
    }
}
```

```
/*****************************************************************************
 * Read a graph from a file into an adjacency list
 * File format: first line: number of vertices (N), number of edges (E)
 *              next N lines: names of N vertices
 *              next E lines: edges - each specified by tail vertex name,
 *                            head vertex name.
 * This file format and code can be used for either an undirected graph or a
 * digraph - see line marked with ** below.  It can also be used for a network
 * if we add a weight to each edge line - see lines marked with **** below
 *****************************************************************************/

GraphL::GraphL(istream & source)
{
    // Read number of vertices and number of edges

    int numberOfEdges;

    source >> _numberOfVertices >> numberOfEdges;

    // Read in names of vertices.  Add a Vertex object for each to the
    // _vertex array, and also temporarily store its name in a map to
    // facilitate reading in the edge descriptions by vertex name

    _vertex = new Vertex[_numberOfVertices];
    map < string, int > nameMap;

    for (int i = 0; i < _numberOfVertices; i ++)
    {
        string name;
        source >> name;
        _vertex[i]._name = name;
        nameMap[name] = i;
    }

    // Read in edges.  Add an edge node for each to the adjacency
    // list for its vertices

    for (int i = 0; i < numberOfEdges; i ++)
    {
        string tail, head;
        source >> tail >> head;
        int tailIndex = nameMap[tail];
        int headIndex = nameMap[head];

        // **** FOR A NETWORK, ADD THE FOLLOWING LINES, AND ADD weight AS A ****
        // **** PARAMETER TO THE Edge CONSTRUCTOR IN THE SUBSEQUENT LINE(S) ****
        // double weight;
        // source >> weight;

        _vertex [ tailIndex ]._edges.push_back(Edge(headIndex));
        // ** THE FOLLOWING LINE WOULD BE OMITTED FOR A DIGRAPH **
        _vertex [ headIndex ]._edges.push_back(Edge(tailIndex));
    }
}
```

```
/*****************************************************************************
 * DFS on an adjacency matrix
 *****************************************************************************/
// The most straight-forward implementation uses a recursive auxiliary
void GraphM::dfs(int start, ostream & output) const  // RECURSIVE VERSION
{
    // Keep track of whether a vertex has been visited, lest we get into a loop
    bool visited [ _numberOfVertices ];
    for (int i = 0; i < _numberOfVertices; i ++)
        visited[i] = false;
    dfsAux(start, output, visited);
}
void GraphM::dfsAux(int current, ostream & output, bool visited []) const
{
       visited[current] = true;
       output << _vertexName[ current ] << endl;
       // Do a DFS recursively from each of its neighbors
       for (int head = _numberOfVertices; head >= 0; head --)
       if (_edge[current] [head] && ! visited[head])
       {
              dfsAux(head, output, visited);
       }
}

// It is also possible to do a non-recursive version using a stack - though
// getting it right is tricky.
void GraphM::dfs(int start, ostream & output) const  // NON-RECURSIVE VERSION
{
    // Keep track of whether a vertex has been visited, lest we get into a loop
    // Note that a vertex can get on the stack several times, but will only
    // be visited once (the first time it is popped  = last time pushed)
    bool visited [ _numberOfVertices ];
    for (int i = 0; i < _numberOfVertices; i ++)
        visited[i] = false;
    stack < int > toVisit;
    toVisit.push(start);
    while (! toVisit.empty())
    {
        // Visit top vertex on the stack - unless we visited it earlier
        int current  = toVisit.top(); toVisit.pop();
        if (! visited[current])
        {
             visited[current] = true;
            output << _vertexName[ current ] << endl;
            // Push its neighbors
            for (int head = _numberOfVertices; head >= 0; head --)
            if (_edge[current] [head] && ! visited[head])
            {
                toVisit.push(head);
            }
        }
    }
}
```

```
/*****************************************************************************
 * BFS on an adjacency list
 *****************************************************************************/

void GraphL::bfs(int start, ostream & output) const
{
    // Keep track of whether a vertex has been scheduled to be visited, lest
    // we get into a loop

    bool scheduled [ _numberOfVertices ];
    for (int i = 0; i < _numberOfVertices; i ++)
        scheduled[i] = false;

    queue < int > toVisit;
    toVisit.push(start);
    scheduled[start] = true;

    while (! toVisit.empty())
    {
        // Visit front vertex on the queue

        int current  = toVisit.front(); toVisit.pop();
        output << _vertex [ current ]._name << endl;

        // Enqueue its unscheduled neighbors

        for (Vertex::EdgeList::iterator neighbor =
                        _vertex[current]._edges.begin();
             neighbor != _vertex[current]._edges.end();
             neighbor ++)
        {
            int head = neighbor -> _head;
            if (! scheduled[head])
            {
                toVisit.push(head);
                scheduled[head] = true;
            }
        }
    }
}

/*****************************************************************************
 * Warshall's transitive closure algorithm - using an adjacency matrix
 *****************************************************************************/

void GraphM::closure()
{
    for (int i = 0; i < _numberOfVertices; i ++)
        for (int j = 0; j < _numberOfVertices; j ++)
            for (int k = 0; k < _numberOfVertices; k ++)
                if (_edge [j] [i] && _edge [i] [k])
                    _edge [j] [k] = true;
}
```

```
/*****************************************************************************
 * Topological sorting - using an adjacency list (two versions)
 *****************************************************************************/

/* Less efficient version:

void GraphL::topsort(ostream & output) const
{
    // Record two facts for every vertex: whether it has been visited, and
    // the count of predecessors that have not yet been visited.  The
    // latter is initialized to zero, then 1 is added for the head of
    // every edge

    bool visited [ _numberOfVertices ];
    int unvisitedPredecessors [ _numberOfVertices ];

    for (int i = 0; i < _numberOfVertices; i ++)
    {
        visited[i] = false;
        unvisitedPredecessors[i] = 0;
    }

    for (int i = 0; i < _numberOfVertices; i ++)
    {
        for (Vertex::EdgeList::iterator iter = _vertex[i]._edges.begin();
                 iter != _vertex[i]._edges.end();
                 iter ++)
            unvisitedPredecessors [ iter -> _head ] ++;
    }


    // Output the vertices in topological order.  The following loop must
    // be done n times if all vertices are visited

    for (int i = 0; i < _numberOfVertices; i ++)
    {
        // Find an unvisited vertex with no unvisited predecessors.  (If
        // none can be found, graph contains a cycle.)

        bool found = false;
        int current;
        for (current = 0; ! found && current < _numberOfVertices; )
            if (! visited[current] && unvisitedPredecessors[current] == 0)
                found = true;
            else
                current ++;

        if (! found)
        {
            output << "Graph contains a cycle - topological sort impossible"
                   << endl;
            return;
        }
```

```
        // Visit the vertex

        output << _vertex[current]._name << endl;
        visited[current] = true;

        // Reduce predecessor count for its successors

        for (Vertex::EdgeList::iterator iter = _vertex[current]._edges.begin();
                iter != _vertex[current]._edges.end();
                iter ++)
            unvisitedPredecessors[ iter -> _head ] --;
    }
}
*/

/* More efficient version */

void GraphL::topsort(ostream & output) const
{
    // Record two facts for every vertex: whether it has been visited, and
    // the count of predecessors that have not yet been visited.  The
    // latter is initialized to zero, then 1 is added for the head of
    // every edge

    bool visited [ _numberOfVertices ];
    int unvisitedPredecessors [ _numberOfVertices ];

    for (int i = 0; i < _numberOfVertices; i ++)
    {
        visited[i] = false;
        unvisitedPredecessors[i] = 0;
    }

    for (int i = 0; i < _numberOfVertices; i ++)
    {
        for (Vertex::EdgeList::iterator iter = _vertex[i]._edges.begin();
                    iter != _vertex[i]._edges.end();
                    iter ++)
            unvisitedPredecessors [ iter -> _head ] ++;
    }

    // Use a queue of visitable vertices - ones that have no unvisited
    // predecessors.  Initially, this queue contains all vertices that
    // have not predecessors in the initial graph

    queue < int > visitable;
    for (int i = 0; i < _numberOfVertices; i ++)
        if (unvisitedPredecessors[i] == 0)
            visitable.push(i);
```

```
    // Output the vertices in topological order.  The following loop must
    // be done n times if all vertices are visited

    for (int i = 0; i < _numberOfVertices; i ++)
    {
        // Find an unvisited vertex with no unvisited predecessors.  (If
        // none can be found, graph contains a cycle.)

        int current;
        if (! visitable.empty())
        {
            current = visitable.front();
            visitable.pop();
        }
        else
        {
            output << "Graph contains a cycle - topological sort impossible"
                   << endl;
            return;
        }

        // Visit the vertex

        output << _vertex[current]._name << endl;
        visited[current] = true;

        // Reduce predecessor count for its successors.  If any drops to
        // zero, add it to the visitable queue

        for (Vertex::EdgeList::iterator iter = _vertex[current]._edges.begin();
                iter != _vertex[current]._edges.end();
                iter ++)
        {
            unvisitedPredecessors[ iter -> _head ] --;
            if (unvisitedPredecessors[ iter -> _head ] == 0)
                visitable.push(iter -> _head);
        }
    }
}
```