CI2 Turno 1 Parte prática. Duração: 1h30m

Submit **only a solution files**, as a zip file (include files *.cpp and *.h).
The zip file **must not include folders**. There is no need to include the file tests.cpp.

- The submitted solution will be tested against an additional set of unit tests. The success on the limited set of provided tests does not ensure full grade in each question/problem.
- Depending on the specific problem, there is the possibility of partial credit. This is not the case for the simples 3 first problems.

This practical test consists in a series of problems dealing graphs. The provided base files a template C++ graph class with T instantiated to `int`, as introduced during your TP classes as transcribed below:

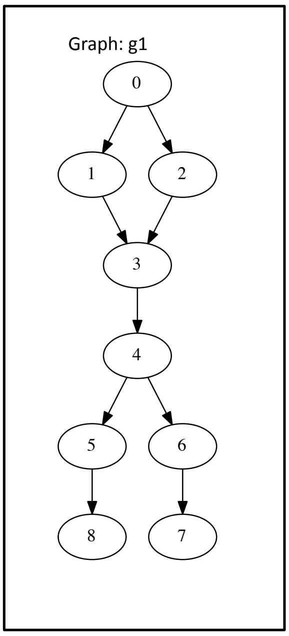
```
#include <cstddef>
#include <vector>
#include <queue>

using namespace std;

template <class T> class Edge;
template <class T> class Graph;
template <class T> class Vertex;


/************ Provided structures ************/

template <class T>
class Vertex {
    T info;                 // contents
    vector<Edge<T> > adj;   // list out edges
    bool visited;           // auxiliary field
    bool processing;        // auxiliary field

    void addEdge(Vertex<T> *dest, double w);
    bool removeEdgeTo(Vertex<T> *d);
public:
    Vertex(T in);
    T getInfo() const;
    void setInfo(T in);
    bool isVisited() const;
    void setVisited(bool v);
    bool isProcessing() const;
    void setProcessing(bool p);
    const vector<Edge<T>> &getAdj() const;
    void setAdj(const vector<Edge<T>> &adj);
    friend class Graph<T>;
};

template <class T>
class Edge {
    Vertex<T> * dest;       // dest vertex
    double weight;          // edge weight
public:
    Edge(Vertex<T> *d, double w);
    Vertex<T> *getDest() const;
    void setDest(Vertex<T> *dest);
    double getWeight() const;
    void setWeight(double weight);
    friend class Graph<T>;
    friend class Vertex<T>;
};

template <class T>
class Graph {
    vector<Vertex<T> *> vertexSet;    // vertex set
    void dfsVisit(Vertex<T> *v, vector<T> & res) const;
    bool dfsIsDAG(Vertex<T> *v) const;
public:
    Vertex<T> *findVertex(const T &in) const;
    int getNumVertex() const;
    bool addVertex(const T &in);
    bool removeVertex(const T &in);
    bool addEdge(const T &sourc, const T &dest, double w);
    bool removeEdge(const T &sourc, const T &dest);
    vector<Vertex<T> * > getVertexSet() const;
    vector<T> dfs() const;
    vector<T> dfs(const T & source) const;
    vector<T> bfs(const T &source) const;
};

/* Auxiliary functions for DFS and BFS */
template <class T>
void Graph<T>::dfsVisit(Vertex<T> *v, vector<T> & res) const {
    v->visited = true;
    res.push_back(v->info);
    for (auto & e : v->adj) {
        auto w = e.dest;
        if ( ! w->visited)
            dfsVisit(w, res);
    }
}

/*
 * Performs a DFS in a graph (this). Returns a vector with the
 * contents of the vertices by DFS order, from the source node. */
template <class T>
vector<T> Graph<T>::dfs(const T & source) const {
   vector<T> res;
   auto s = findVertex(source);
   if (s == nullptr)
    return res;

   for (auto v : vertexSet)
     v->visited = false;

   dfsVisit(s, res);
   return res;
}

/* Performs a BFS in a graph (this), starting at (source).
 * Returns a vector with vertices' contents by DFS order. */
template <class T>
vector<T> Graph<T>::bfs(const T & source) const {
   vector<T> res;
   auto s = findVertex(source);
   if (s == NULL)
     return res;
   queue<Vertex<T> *> q;
   for (auto v : vertexSet)
     v->visited = false;
   q.push(s);
   s->visited = true;
   while (!q.empty()) {
     auto v = q.front();
     q.pop();
     res.push_back(v->info);
     for (auto & e : v->adj) {
      auto w = e.dest;
      if ( ! w->visited ) {
        q.push(w);
        w->visited = true;
      }
     }
   }
   return res;
}
```

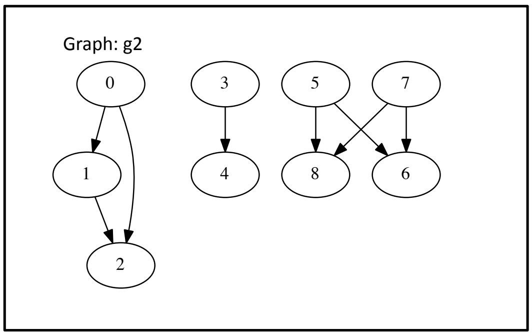
In addition to this base class, we provide an example of an implementation of the Depth-First Search and Breath-First Search traversal implementations (as used in your TP classes). You may use and modify these implementations.

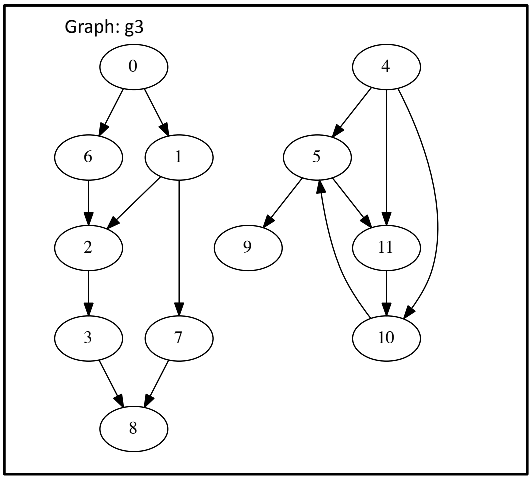
However, do not modify the "construtors" or the functions that add and remove edges or vertices, as these are used to construct the "input" to your code. You may, nevertheless, add new functions and/or variables at the class level.

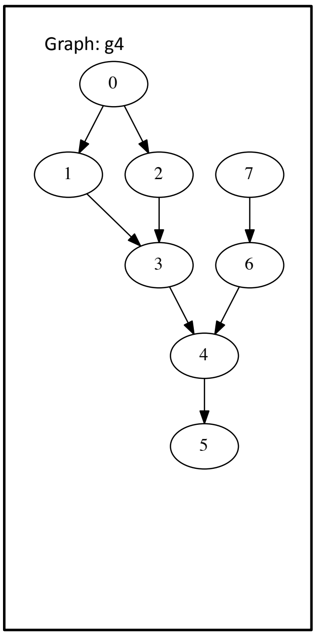
The test programs contains some “pre-defined” test graphs used as part of the test suite of graphs. To simplify your task we have illustrations of some of these graphs below where the `info` field of each node (or `Vertex`) is an integer `int`:

Below are some of the directed and uweighted graphs:









In addition, you are given a class named as ***GraphOps*** (files .h and .cpp) with the indication of the methods you are expected to develop as your solutions to the various problems.

PROBLEMS DESCRIPTION

Problem 1 [3.0] Absence of directed connection between nodes

Implement the following function in the file ***GraphOps.cpp***:

```
bool GraphOps::directlyUnconnected(Graph<int> g, int u, int v)
```

This function returns *true* if there is no **direct** edge between the nodes in the graph whose `info` fields are respectively `u` and `v`, or *false* in case the edge exists. In case one of the nodes `u` or `v` does not exists in the graph, the function should return *true*.

Invocation examples:

```
Graph<int> g1 = graph1();
cout << GraphOps::directlyUnconnected(g1, 1, 2) << endl;
Graph<int> g1 = graph1();
cout << GraphOps::directlyUnconnected(g1, 1, 3) << endl;
Graph<int> g1 = graph1();
cout << GraphOps::directlyUnconnected(g1, 3, 1) << endl;
Graph<int> g4 = graph4();
cout << GraphOps::directlyUnconnected(g4, 2, 3) << endl;
Graph<int> g4 = graph4();
cout << GraphOps::directlyUnconnected(g4, 4, 2) << endl;
```

Expected results:

```
1
0
1
0
1
```

Explanation:

graph1 – the edge (1,2) does not exist, the edge (1,3) exists; the edge (3,1) not exist.
graph4 – the edge (2,3) exists but the edge (4,2) does not exist

Problem 2 [3.0] Largest outbound degree

Implement the following function in the file ***GraphOps.cpp***:

```
vector<int> GraphOps::largestOutDegree(Graph<int> g)
```

This function must return a *vector* with the identifier `info` of all the nodes in the graph that have the maximum outbound degree. The nodes may be inserted in the return *vector* in any order.

Invocation examples:

```
Graph<int> g1 = graph1();
auto v1 = GraphOps::largestOutDegree(g1);
for (auto v : v1) cout << v << " "; cout << endl;
Graph<int> g2 = graph2();
auto v2 = GraphOps::largestOutDegree(g2);
for (auto v : v2) cout << v << " "; cout << endl;
```

Expected results:

```
0 4
0 5 7
```

Explanation:

graph1 – nodes 0 and 4 both have the largest outbound degree, 2 in this case

graph2 – nodes 0 and 5 and 7 all have the largest outbound degree, 2 in this case

Problem 3. [3.0] Absence of cycles (is a DAG)

Implement the following function in the file ***GraphOps.cpp***:

```
bool GraphOps::isDAG(Graph<int> g)
```

This function returns *true* if there are no cycles between any nodes in the graph, and *false* otherwise.

Invocation examples:

```
Graph<int> g1 = graph1();
cout << GraphOps::isDAG(g1) << endl;
Graph<int> g3 = graph3();
cout << GraphOps::isDAG(g3) << endl;
```

Expected results:

```
1
0
```

Explanation:

graph1 *it does not have any cycles*
graph3 *there is cycle with nodes 5, 11 and 10*

Problem 4. [4.0] Sources and Sinks nodes

Implement the following function in the file ***GraphOps.cpp***:

```
vector<int> GraphOps::numberSourcesSinks(Graph<int> g)
```

This function must return *vector* with only 2 integer values, respectively, the number of nodes in the graph that are source nodes (i.e., nodes that are not the destination of any edge) and the number of nodes in the graph that are sink nodes (i.e., nodes that are not the source of any edge). Isolated nodes are both *source* and *sink* nodes.

Invocation examples:

```
Graph g4 = graph4();
cout << auto v1 = GraphOps::numberSourcesSinks(g4) << endl;
for (auto v : v1) cout << v << " "; cout << endl;
```

Expected results:

```
2
1
```

Explanation:
graph4 nodes 0 and 7 have no edges with these nodes are destination - they are source nodes.
nodes 5 have no edges with this node as source - it is a sink node.

Problem 5. [3.0] Paths

Implement the following function in the file ***GraphOps.cpp***:

```
bool GraphOps::pathExists(Graph<int> g,int s,int t,const vector<int> & skip)
```

This function returns *true* if there exists at least one path between the nodes with identifiers s and t which does not go through the nodes with the identifiers in the vector skip, or *false* otherwise. You may assume that s, t and skip have identifiers that correspond to existing nodes in the graph.

Invocation examples:

```
Graph g1 = graph1();
cout << GraphOps::pathExists(g1, 1, 5, {2}) << endl;
Graph g1 = graph1();
cout << GraphOps::pathExists(g1, 1, 5, {3}) << endl;
Graph g1 = graph1();
cout << GraphOps::pathExists(g1, 2, 6, {4}) << endl;
Graph g1 = graph1();
cout << GraphOps::pathExists(g1, 2, 6, {7,8}) << endl;
```

Expected results:

```
1
0
0
1
```

Explanation:

graph1 there exists one path between nodes 1 and 5 that does not go through node 2 (1→3→4→5)
there is no path between nodes 1 and 5 that does not go through node 3
there is no path between nodes 2 and 6 that does not go through node 4
there exists one path between nodes 2 and 6 that does not go through nodes 7 and 8 (2→3→4→6)

Problem 6. [4.0] Connected Components

Implement the following function in the file ***GraphOps.cpp***:

```
int::GraphOps::numberConnectedComponents(Graph<int> g)
```

This function is expected to return the number of connected components in the input graph. We are not interested in determining the actual nodes in each connected component, but simples how many are there in the input graph.

Invocation examples:

```
Graph<int> g2 = graph2();
cout << GraphOps::numberConnectedComponents(g2) << endl;
Graph<int> g3 = graph3();
cout << GraphOps::numberConnectedComponents(g3) << endl;
```

Expected results:

```
3
2
```

Explanation:
graph2 – has 3 connected components, although not strongly connected.
graph3 – has 2 connected components, although not strongly connected.

Submit only solutions files, as a zip file (include files *.cpp and *.h).

The zip file **must not include folders**.

There is no need to include the file tests.cpp.