12th Recitation

Graphs: Topological Sorting and Connected Components

Instructions:

- Download the file `aed2324_p12.zip` from the course page and unzip it (it contains the `lib` folder, the `Tests` folder with the files `Graph.h`, `funWithGraphs.h` , `funWithGraphs.cpp` and `tests.cpp`, and the files `CmakeLists.txt` and `main.cpp`);
- In CLion, open a project by selecting the folder containing the files from the previous point;
- If you can't compile, perform "Reload CMake Project" on the `CMakeLists.txt` file;
- Implement it in the file `Graph.h` and `funWithGraphs.cpp`
- Note that all the tests are uncommented. They should fail when run for the first time (before implementation). As you solve the exercises, the respective tests should pass.

1. Graphs: Topological Sorting

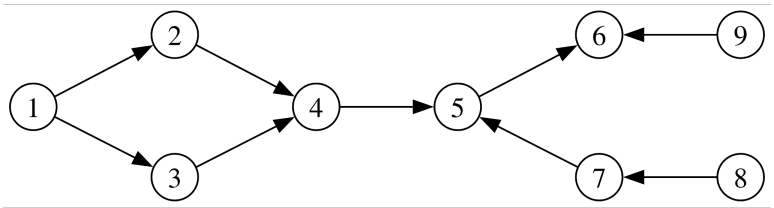
Consider the `Graph` class below, as defined in the `Graph.h` file:

```
template <class T> class Vertex {
    T info;
    vector<Edge<T> > adj;
public:
    //...
    friend class Graph<T>;
};

template <class T> class Edge {
    Vertex<T> * dest;
    double weight;
public:
    //...
    friend class Graph<T>;
    friend class Vertex<T>;
};
```

```
template <class T> class Graph {
    vector<Vertex<T> *> vertexSet;
    //...
public:
    //...
};
```

and the following graph as an example:



Implement the `Graph` class member function below using the algorithm described in class:

```
vector<T> topsort() const
```

This function returns a vector containing the elements of the graph (member-data info of the vertices) ordered topologically. When a topological sort is not possible for the graph in question, that is, when it is not a DAG, the vector to be returned should be empty.

Execution example. Some possible topological orders for the graph depicted above are listed below:

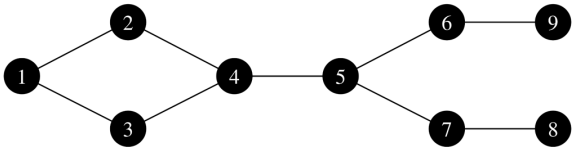
output: *result* = `{1, 2, 3, 4, 8, 7, 9, 5, 6}`, or
= `{1, 8, 9, 2, 3, 7, 4, 5, 6}`, or
= `{8, 9, 7, 1, 3, 2, 4, 5, 6}, ...`

---

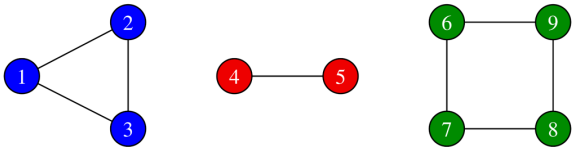
2. Graphs: Connected Components and Articulation Points

2.1 Connected Components (*number of*)

Recall that a connected component of an undirected graph is a subgraph in which any pair of vertices has a path between them. For example, *Graph 1* (below, left) has only one connected component (indicated in black), while *Graph 2* (below, right) has 3 connected components indicated in blue, red and green.



Graph 1



Graph 2

To count the number of connected components you are asked to implement the member function below in the `funWithGraphs.cpp` file:

```
int connectedComponents(Graph<int> *g)
```

This function returns the number of connected components of the graph (assume that it is an undirected graph). As the base Graph class supports directed graphs only, you may "convert" it to an undirected graph by duplicating all the edges swapping the source and destination end points. Note that this "artifact" only works for this specific problem as in other cases the obvious cycles it introduces, will render simple graph-based algorithms ineffective.

Execution example, for the graphs depicted above:

input: *graph 1*
output: *result* = `1`

input: *graph 2*
output: *result* = `3`

Suggestion: Perform a depth-first search (DFS) from each node not yet visited and count how many times a new search is started.

2.2 Giant Component.

Implement the following function in the `funWithGraphs.cpp` file:

```
int giantComponents(Graph<int> *g)
```

This function should return the size (number of vertices) of the largest connected component of the graph, i.e., the one that has the largest number of vertices.

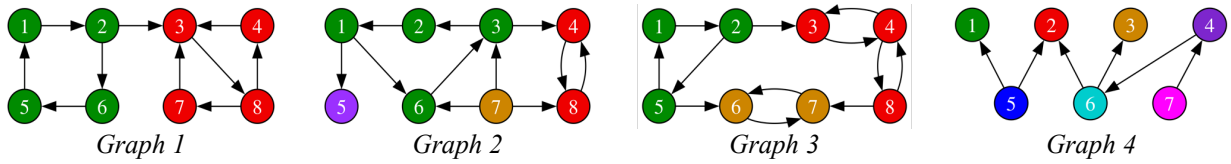
Execution example, for the graphs depicted above:

input: *graph 1*	input: *graph 2*
output: *result* = `9`	output: *result* = `4`
Has only one component of 9 vertices;	Has 3 components of sizes 3, 2 and **4**.

Suggestion: Extend the code from the previous exercise to also count the vertices; or change the DFS to return the number of vertices in the component.

2.3 Strongly Connected Components

Remember that a strongly connected component of a directed graph is a maximal subgraph in which any pair of vertices has a path between them. The figure below illustrates several graphs and their respective strongly connected components (indicated by the colors).



Graph 1 *Graph 2* *Graph 3* *Graph 4*

- *Graph 1* has **two** strongly connected components: `{1, 2, 5, 6}` and `{3, 4, 7, 8}`
- *Graph 2* has **four** strongly connected components: `{1, 2, 3, 6}`, `{4, 8}`, `{5}` and `{7}`
- *Graph 3* has **three** strongly connected components: `{1, 2, 5}`, `{3, 4, 8}` and `{6, 7}`
- *Graph 4* has **seven** strongly connected components: `{1}`, `{2}`, `{3}`, `{4}`, `{5}`, `{6}` and `{7}`

To list strongly connected components, implement the following function in the `funWithGraphs.cpp` file:

```
list<list<int>> scc(Graph<int> *g)
```

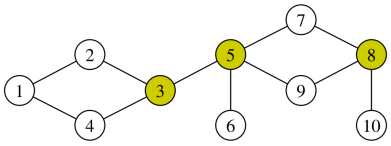
This function returns a list of lists of integers containing the strongly connected components of the input graph `g`.

Suggestion: Use Tarjan's algorithm, as presented in class. Consider the auxiliary function below, as well as the `processing`, `num` and `low` attributes, already defined in the `Graph` class.

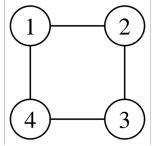
```
void dfs_scc(const Graph<int> *g, Vertex<int> *v, stack<int> &s, list<list<int>> &l, int &i);
```

2.4 Articulation Points

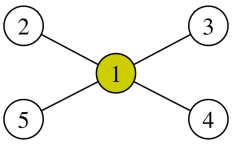
Remember that an articulation point in an undirected graph is a vertex which, if removed, increases the number of connected components. The figure below illustrates several graphs and their respective articulation points (indicated in yellow).



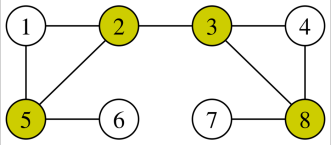
Graph 1



Graph 2



Graph 3



Graph 4

- *Graph 1* has **three** articulation points: vertices `3`, `5` and `8`
- *Graph 2* has no articulation points
- *Graph 3* has **one** articulation point: node `1`
- *Graph 4* has **four** articulation points: nodes `2`, `3`, `5` and `8`

Implement the following function in the `funWithGraphs.cpp` file:

```
unordered_set<int> articulationPoints(Graph<int> *g)
```

Suggestion: Use the algorithm presented in the lectures, based on Trajan's algorithm. Consider the auxiliary function below, as well as the `processing`, `num` and `low` attributes, already implemented in the `Graph` class.

```
void dfs_art(Graph<int> *g, Vertex<int> *v, stack<int> &s, unordered_set<int> &res, int &i);
```