4th Practical Class – Graphs: representation, depth-first and breadth-first search, topological sort

Instructions

- Download the zipped file **cal_fp04_CLion.zip** from the course's Moodle area and unzip it (it contains the folder **lib,** the folder **Tests** with files **tests.cpp**, **Graph.h**, **Person.h**, **Person.cpp**, and the files **CMakeLists** and **main.cpp**.
- In the CLion IDE, open a project, selecting the folder that contains the files mentioned in the previous bullet point.
- Do "*Load CMake Project*" over the file *CMakeLists.txt*
- Run the project (**Run**)
- Please note that the unity tests of this project may be commented. If this is the case, uncomment the tests as you make progress in the implementation of the respective exercises.
- You should implement the exercises following the order suggested.
- Implement your solutions in the respective .cpp file of each .h. Remember that template classes should be implemented in the .h files!
- Important note: in case you need to read text files in I/O mode, you should tell CLion where such files are, by redefining the IDE environment variable "Working Directory", through menu Run > Edit Configurations… > Working Direcory
- The code to be completed in the **Graph.h** file is marked with **TODO** and accompanied by explanatory comments and tips.

Exercises

1. Graphs: representation and CRUD

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:
	…
};
```

a) In the **Graph** class, implement the member function below:
   `bool addVertex(const T &in)`

   This function adds the vertex with content **in** (info) to the graph. It returns **true** if the vertex was added successfully and **false** if one with the same content already exists.

b) In the **Graph** class, implement the member function below:
   `bool addEdge(const T &sourc, const T &dest, double w)`

   This function adds to the graph the edge originating at vertex **sourc**, ending at vertex **dest**, and weight **w**. It returns **true** if the edge was added successfully and **false** if it is not possible to insert that edge (because the vertices with the indicated content do not exist).

c) In the **Graph** class, implement the member function below:
   `bool removeEdge(const T &sourc, const T &dest)`

   This function removes the edge originating at vertex **sourc** and ending at vertex **dest**. It returns **true** if the edge was removed successfully and **false** otherwise (edge does not exist).

d) In the **Graph** class, implement the member function below:
   `bool removeVertex(const T &in)`

   This function removes the vertex with content **in**. It returns **true** if the vertex was removed successfully and **false** otherwise (the vertex does not exist). Removing a vertex implies removing all edges with origin and / or destination at that vertex.

2. Graphs: Depth-first and breadth-first search; topological sort

a) In the **Graph** class, implement the member function below according to the algorithms presented in the theory classes:
   `vector<T> dfs() const`

   This function returns a vector containing the graph elements (content of the vertices) when an depth-first search is performed on the graph.

b) In the **Graph** class, implement the member function below according to the algorithms presented in the theory classes:

   `vector<T> bfs(Vertex<T> *s) const`

   This function returns a vector containing the graph elements (content of the vertices) when an breadth-first search is performed on the graph, starting at vertex **s**.

e) In the **Graph** class, implement the member function below according to the algorithms presented in the theory classes:

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

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 will be empty.

3. Practical applications of depth-first and breadth-first algorithms (Homework)

a) A social network can be represented by the **Graph** class, where the vertices represent individuals in the network and the edges represent a friendship relationship (the direction of the edge indicates who has sent the friend request).

After learning of important news, an individual sends it out through his/her friendships. Determine which individual (vertex) disseminates the news first hand (as a novelty) to the largest number of recipients and what is the number of recipients reached.

In the **Graph** class, implement the member function below:

```
int maxNewChildren(Vertex<T> *v, T &inf) const
```

This function returns the maximum number of children in a graph starting at vertex **v**; the **inf** parameter is the content of that graph vertex. Tip: adapt the breadth-first search algorithm.

b) In the **Graph** class, implement the member function below:

```
bool isDAG()
```

This function checks whether the directed graph is acyclic (Directed Acyclic Graph, or DAG for short), that is, the graph does not contain cycles, in which case the function returns **true**. Otherwise, the function returns **false**. Tip: Adapt the depth-first search algorithm.