Review of Simple Graphs Algorithms & Strongly-Connected Components Algorithms

Practical Exercises

Departamento de Engenharia Informática (DEI)
Faculdade de Engenharia da Universidade do Porto (FEUP)

Spring 2025

We recommend you to use CLion. Make sure you have CLion installed and a **valid student license** (read CLion Installation tutorial from moodle). After that, open the *da2425_p01_student* folder using CLion. The following exercises use the **Graph.h** provided in the *da2425_p01_student/data_structures* directory, which includes the basic functionalities and includes all the necessary attributes for the **Vertex**, **Edge,** and **Graph**. Analyze the structure carefully before starting solving the exercises. There are auxiliary attributes and methods that will only be useful in future lessons.

Start by running the program. All tests should fail (with **red** color) because there is no valid implementation of the functions yet. All unit tests must pass (with **green** color) after the methods have been correctly implemented.

Exercise 1

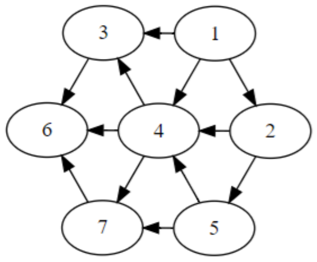
In the **ex1.cpp** file, implement the following function to create a vector containing the IDs of graph nodes in DFS order, starting from the source node. Use the `getInfo()` method to retrieve the node IDs and utilize the other auxiliary methods, such as `findVertex(const T &source)`, `getVertexSet()`, `getAdj()`, `getDest()`, and `setVisited(bool visited)` as needed.

```
vector<T> dfs(Graph<T> *g, const T & source)
```

Additionally, in the **ex1.cpp** file, implement the function to create a vector containing the IDs of graph nodes in BFS order, starting from the source node.

```
vector<T> bfs(const Graph<T> *g, const T & source)
```

Exercise 2
Consider the directed and acyclic graph (DAG) below:



(a) Indicate two different topological sorts of its vertices.

(b) In the **ex2.cpp** file, implement the function below:

```
vector <T> topsort(Graph<T>* g)
```

This function returns a vector containing the IDs of the graph nodes ordered topologically. When a topological sort is not possible for the graph, that is, when it is not a DAG, the vector to be returned will be empty.

Suggestion: Use the *indegree* attribute (and associated getter and setter) from the **Vertex** class.

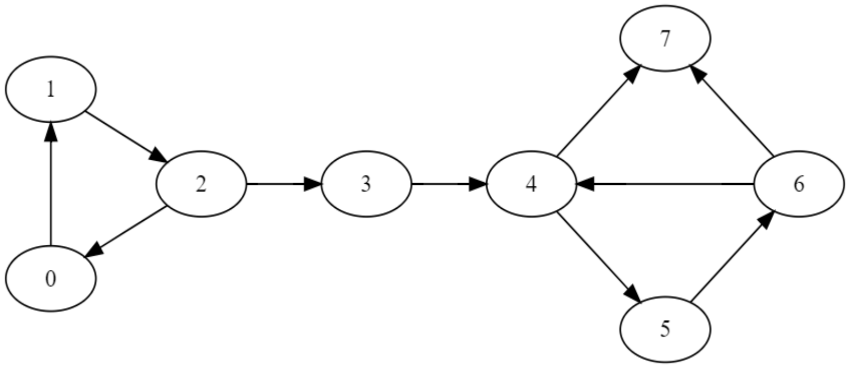
Exercise 3
In the **ex3.cpp** file, implement the function below.

```
bool isDAG(Graph<T> *g)
```

This function determines whether a directed graph is DAG, that is, it does not contain cycles.

Suggestions: Adapt the Depth-First Search algorithm using the *visited* and *processing* attributes (and associated getters and setters) from the **Vertex** class.

Exercise 4
Consider the directed graph G depicted below.



For this graph G:

a) Compute G's Strongly Connected Components (SCCs). An SCC is a subset of a directed graph's vertices, where, from any vertex, it is possible to reach any other vertex.

b) In the **ex4.cpp** file, implement a function that computes a directed graph's SCCs using the Kosaraju-Sharir algorithm. The function should return a vector of SCCs, where each SCC is represented as a vector containing the IDs of its vertices.

```
vector<vector<T>> SCCkosaraju (Graph<T> *g)
```

Suggestion

1) Perform DFS on the original graph.
2) Use a Stack and push nodes to stack before returning the recursive call.
3) Find the transpose graph by reversing the edges.
4) Pop nodes one by one from the stack and again run the DFS on the modified graph.

Exercise 5

In the **ex5.cpp** file, implement a function that computes a directed graph's SCCs using Tarjan's algorithm. The function should return a vector of SCCs, where each SCC is represented as a vector containing the IDs of its vertices.

```
vector<vector<T>> sccTarjan(Graph<T>* g)
```

Recall the algorithm:

Tarjan Algorithm for SCCs

```
index ← 1 ; $S \leftarrow \emptyset$
For all nodes $v$ of the graph do
  If $num[v]$ is still undefined then
    dfs_scc($v$)

dfs_scc(node $v$):
  $num[v] \leftarrow low[v] \leftarrow index$ ; $index \leftarrow index + 1$ ; S.push($v$)
  /* Traverse edges of $v$ */
  For all neighbors $w$ of $v$ do
    If $num[w]$ is still undefined then /* Tree Edge */
      dfs_scc($w$) ; $low[v] \leftarrow min(low[v], low[w])$
    Else if $w$ is in $S$ then /* Back Edge */
      $low[v] \leftarrow min(low[v], num[w])$
  /* We know that we are at the root of an SCC */
  If $num[v] = low[v]$ then
    Start new SCC $C$
    Repeat
      $w \leftarrow$ S.pop() ; Add $w$ to $C$
    Until $w = v$
    Write $C$
```

Extra

You can uncomment the final unit test (*test_extratests)* and experiment the implemented functions with multiple different graphs created in createGraphs.cpp file. The graphs can be visualized in SampleGraphs.pdf.