Review of Simple Graphs Algorithms and Strongly-Connected Components Algorithms

Practical Exercises

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

Spring 2024

**Various of these problems were originally developed by the team of instructors of the 2021 CAL (Concepção de Algoritmos") course of the L.EIC at the Faculty of Engineering of the University of Porto (FEUP), and adapted for use in this class.*

A - Graphs

Exercise 1
The following exercises, use the **Graph** class provided in the *data_structures* directory. **This class** provides the basic functionalities and includes all the necessary attributes for the **Vertex**, **Edge,** and **Graph**.

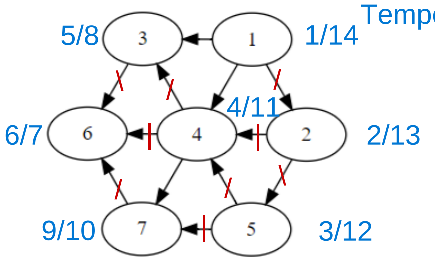
The base **Graph** class represents the graph as an adjacency list and already has methods to add vertices and edges to the graph. In this exercise, methods will be created to remove edges and vertices from the graph.

Given the directed and acyclic graph (DAG) below:

a) Indicate two different topological sorts of its vertices.

Time(global) = 1

Tempo chegada / Tempo finalização



Lista com resultados : (1) (2) (5) (4) (7) (3) (6)

b) In the **ex1.cpp** file, implement the function below:

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

This function returns a vector containing the elements of the graph (in this case the identifiers 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.

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

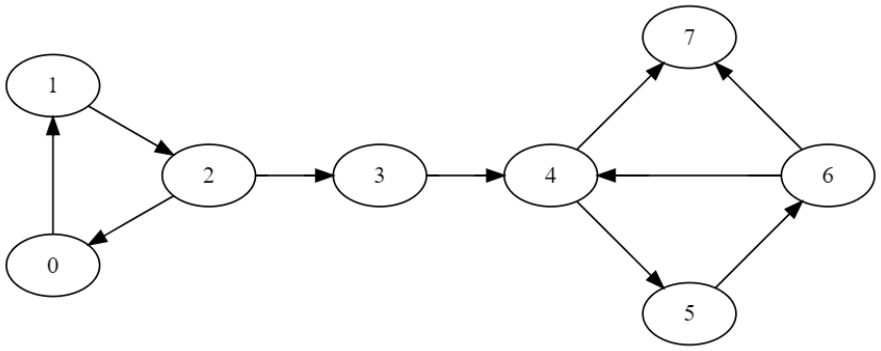
Exercise 2
In the **ex2.cpp** file, implement the function below.

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

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 3
Consider directed graph G depicted below.



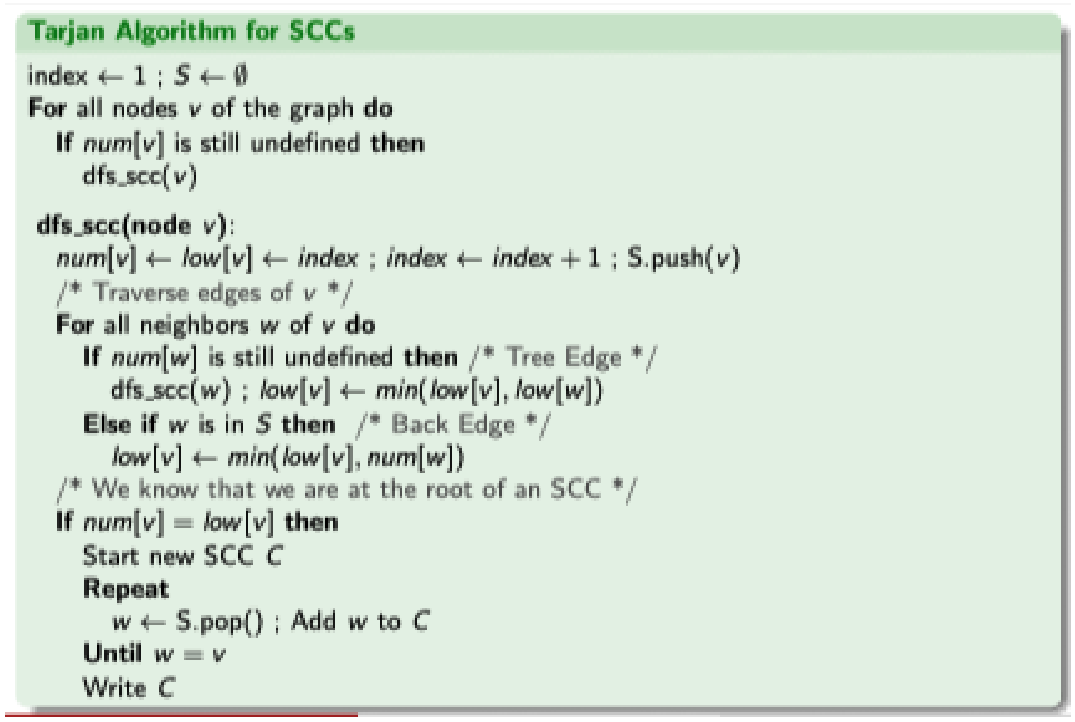
For this graph G:

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 **ex3.cpp** file, implement a function that computes a directed graph's SCCs using the Kosaraju-Sharir algorithm.

```
vector<vector<T>> SCCkosaraju (const 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.

```
list<list<T>> sccTarjan(const Graph<T>* g)
```



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 $C$
```