Exercise 3

Introduction

The third exercise requires to design and implement the DFS visit without an auxiliary data structure, in order to have an iterative algorithm. The DFS (depth-first-search) is a search algorithm on trees and graphs that even before having visited the nodes of the first generations can find itself visiting vertices far from the root, thus going "in depth".

We provide two solution of this problem that differ only in the way the two functions take note of the discovered vertices.

We created a new class called “ExtendedGraph” that extends “Graph” class, in which we inserted the nested class “ExtendedVertex” that extends “Vertex” class, that has two new attributes:

- Predecessor, that is the reference to the node that preceded it in the DFS.
- Discovered, that takes note if the vertex has been discovered or not.

These extended classes provide also methods for setting and getting this new attribute, print the graph and it overrides the default insert_vertex function in order to use the new ExtendedVertex class.

First solution.

In the first solution we call the DFS_complete_iterative function that creates a dictionary called “forest” that will contains all the couple (vertex, edge used to discover it). In the DFS_iterative there is a “while” loop that ends when the current vertex is *None* (i.e. the parent of the root). If the vertex is not *None*, the algorithm checks if the degree of the vertex is greater than zero (otherwise it sets its predecessor as current vertex). If the degree is greater than zero, for all the incident edges of the current vertex (until the first undiscovered vertex is found) and if the other endpoint is not been discovered yet (i.e. it has the attribute discovered set to *False*), it sets the discovered attribute to *True* by the method “set_discovered”, sets its predecessor by the method “set_predecessor” and sets it as the new current vertex. In order to return on the previous vertex we use the method “get_predecessor” which returns the reference to the parent of the current vertex, so in this way we can go back to the root in iterative form and without using a stack.

Second solution.

In the dfs_iterative.py file there is also a second solution (the DFS_iterative_with_discovered function), that uses the “forest” dictionary also to track all the discovered vertex until that moment, instead of the discovered attribute of the Vertex class.

Please note: the second solution was implemented only to respect the prototype of the recursive DFS function of the TDP_collection library, but the logic of the algorithm is the same in both solutions.

Complexity analysis

In both solutions the time complexity are the same of the classic DFS algorithm (**O(n+m)** where m is the number of edges and n is the number of the vertices), but the spatial complexity changes because in the first solution we use one variable in addition, while in the second solution we use two additional variables.