Problem3

Solution

In order to implement the DFS algorithm iteratively, we have changed the structure of the Vertex class. We have added two variables:

1. **dfs_predecessor**, that represents the vertex from which we reached the current one, set by default to None.
2. **edges_to_visit**, that memorizes the iterator of the incident edges of the current vertex, set by default to None.

The implementation is composed by two functions, **dfs_complete(g)** and **dfs(g,s)** that are in the **iterative_dfs.py** module.

The **dfs_complete(g)** function takes as input a graph, instance of the Graph class, present in the graph.py module.
It looks for each not yet visited connected component of the graph received as an argument, and calls the *dfs(g,s)* function on each of these connected components. If the graph is connected, this results in only one call to the *dfs(g,s)* function.

The **dfs(g,s)** function takes as input the graph and a vertex representing the starting point of the DFS visit (the *source*).
Starting from the source, the visit consists in saving the iterator for the incident edges in *edges_to_visit*, setting the *dfs_predecessor* to None, and considering the vertices adjacent to it.
For each adjacent vertex, the algorithm checks if it has already been visited, by inspecting the value of the *edges_to_visit* attribute (finding it set to None means the vertex has not been visited yet), and, if it's not the case, it sets the source as its *dfs_predecessor*, and makes this vertex the current one. The algorithm then proceeds as for the source.
When all the vertices adjacent to a certain one have been visited, the algorithm "goes back" to the *dfs_predecessor* of the current node, by setting it as the new current one.
The algorithm terminates when the predecessor of the current node is None, because this indicates that this node is the source of the visit.

Time complexity

The time complexity of the algorithm is the same of the recursive one, $O(n+m)$.
We can prove this by analizing separately the two functions in the **iterative_dfs.py** module.
The **dfs_complete(g)** function performs a constant-time if statement on every node of the graph `g`, and a call to **dfs(g, s)**, on a certain source node, for each of its connected component.
In the **dfs(g, s)** function, the inner while performs constant-time operations on each of the edges adjacent to a current one (named *s*), and it's exectued, toghether with a constant-time if statement, for each vertex in the connected component of `g` that contains `s` (outer while). So this functions'

complexity is $O(\sum_{v \in G(s)} deg(v))$, where *G(s)* is the connected component of `g` that contains `s`. Since this function is called by **dfs_complete(g)** on each of the connected components of `g`, and the connected components are disjoint sets of vertices, this results in a total complexity of $O(\sum_{G(s) \in g}(\sum_{v \in G(s)} deg(v)))$, that is $O(\sum_{v \in g} deg(v))$, that is $O(n + m)$.

Alternative solutions

We have considered two different solutions in addition to the proposed one.
These are illustrated below.
For each of them, we report only the implementation of the **dfs(g,s)** function, because it's the core function of the algorithm.

The first alternative solution doesn't save the iterator as an attribute of the Node instance, but still adds two instance memeber to the class: *dfs_predecessor*, with the same meaning of the chosen implementation, and *visited*, that stores a boolean value, indicating wheter the node has been already visited or not.
The Python implementation of the **dfs(g,s)** function for this alternative solution is reported below:

```
def dfs(g, s):
    # visit the root of the DFS tree
    s.visited = True
    # the root has no predecessor
    s.dfs_predecessor = None

    while True:
        # Get an iterator over the incident edges in the node
        edges_to_visit = g.incident_edges(s)
        while True:
            try:
                e = next(edges_to_visit)
            except StopIteration:
                break
            v = e.opposite(s)
            if not v.visited:
                # Visit the node on the opposite side of the edge
                v.visited = True
                v.dfs_predecessor = s
                # Move on it
                s = v
                edges_to_visit = g.incident_edges(s)

        if s.dfs_predecessor is not None:
            # go back to the previous node in the DFS tree, to continue
            # the inspection of its incident edges
```

```
            s = s.dfs_predecessor
        else:
            # all the incident edges for the root have been "relaxed",
            # so all the nodes which are reachable from the root have
            # have been visited, hence the algorithm can stop
            break
```

We didn't choose this as our final solution, because the fact that we don't save the iterator implies that, every time we go back to a previously visited node, to inspect its adjacent vertices, we consider again all the edges incident to it, and this results in quadratic complexity on the single vertex, and this brings to poorer perfomances, in terms of time complexity, with respect to the chosen solution. In fact, the first edge in the iteration is considered $deg(v)$ times, the second $deg(v) - 1$, the third $deg(v) - 2$ and so on. So, for a single vertex, we have a complexity that is $O(deg(v)^2)$, and this results in a total complexity on the connected component of $O(\sum_{v \in G(s)} deg(v)^2)$, where *G(s)* is the connected component of `g` that contains `s`. By calling this function on each of the connected components of graph `g`, we obtain a time complexity on the whole graph that is $\Omega(n + m)$.

The second alternative solution instead modifies the implementation of the **incident_edges(self, v, outgoing=True)** method in the Graph class, in order to make it return a list of the edges incident to the vertex `v`, in the way shown below:

```
    def incident_edges(self, v, outgoing=True):
        """Return all (outgoing) edges incident to vertex v in the graph.

        If graph is directed, optional parameter used to request incoming
edges.
        """
        self._validate_vertex(v)
        adj = self._outgoing if outgoing else self._incoming
        # Need to explicitly create a list, because in Python3
dictionary.values() returns a dict_values object,
        # that is not indexable.
        return list(adj[v].values())
```

Moreover, to implement this solution, we have added to the Graph.Vertex class two members, *dfs_predecessor*, that we have already specified, and *edges_to_visit*, which, in this implementation, represents the index, in the list returned by *graph.incident_edges(v)*, of the next edge that we have to "relaxe", for that node. This solves the problem of considering several times edges that we had already considered, but it's still inefficient, because the creation of the incident edges list, in the way shown above, has complexity $O(deg(v))$, where v is the current vertex, and this operation is repeated each time we come back to this vertex from an adjacent one, that, in the worst case, is $O(deg(v))$ times, so this algorithm has complexity similar to the first alternative, and that's why we haven't chosen this as our definitive solution.

Additional considerations

The provided algorithm performs the same conceptual operation of the recursive one, that is, identify, for each vertex of a given connected component of the given graph, its predecessor in the DFS tree, and mark the predecessor of the source vertex of the DFS visit as None. To build a graph representing the DFS tree of one of the connected components, meant as and additional Graph class instance, some additional computation is necessary. This computation consists in creating a new graph, and fullfil it with the vertices of the connected component, linked according to the information contained in the dfs_predecessor field of the Node instances.
The function implementing this operation is not provided, since it wasn't part of the recursive DFS implementation neither. However, this does not causes our time complexity analysis to be incorrect, since this additional computation would require time $O(n)$, that would be additional with respect to the time required by the DFS algorithm, so the overall complexity would still be $O(n + m)$.

Usage instructions

To use the algorithm, it's only necessary to call the dfs_complete function, in the **iterative_dfs.py** giving as an argument the desired graph, with the only constraint that this graph is an instance of the class given in the **graph.py** file.
The **main.py** module builds a trivial graph and calls the algorithm on it.