Solution Review: Find a "Mother Vertex" in a Directed Graph

This review provides a detailed analysis of the different ways to solve the find a "mother vertex" in a graph challenge.

We'll cover the following

Solution #1: Naive solution#

main.py

Graph.py

Stack.py

Queue.py

LinkedList.py

Node.py

```
from Graph import Graph
from Stack import MyStack
# We only need Graph and Stack for this question!

```

```
def find_mother_vertex(g):
    # Traverse the Graph Array and perform DFS operati
    # The vertex whose DFS Traversal results is equal to the total nu
    # of vertices in graph is a mother vertex
    num_of_vertices_reached = 0
    for i in range(g.vertices):
        num_of_vertices_reached = perform_DFS(g, i)
        if (num_of_vertices_reached is g.vertices):
            return i
    return - 1

    # Performs DFS Traversal on graph starting from source
    # Returns total number of vertices which can be reached from sour


def perform_DFS(g, source):
    num_of_vertices = g.vertices
    vertices_reached = 0  # To store how many vertices reached from s
    # A list to hold the history of visited nodes (by default all fal
    # Make a node visited whenever you push it into stack
    visited = [False] * num_of_vertices
    # Create Stack (Implemented in previous section) for Depth First
    # and Push source in it
    stack = MyStack()
```

This is the brute force approach for solving this problem. We run a DFS on each vertex using `perform_DFS` and keep track of the number of vertices visited in the search. If it is equal to `g.vertices`, then that vertex can reach all the vertices and is, hence, a mother vertex.

The algorithm would also work with a BFS using a queue.

Time complexity#

Since we run a DFS on each node, the time complexity is $O(V(V + E))$

Solution #2: Last finished vertex#

main.py

Graph.py

Stack.py

Queue.py

LinkedList.py

Node.py

```
from Graph import Graph
from Stack import MyStack
# We only need Graph and Stack for this question!


def find_mother_vertex(g):
    # visited[] is used for DFS. Initially all are
    # initialized as not visited
    visited = [False]*(g.vertices)
    # To store last finished vertex (or mother vertex)
    last_v = 0
    # Do a DFS traversal and find the last finished
    # vertex
    for i in range(g.vertices):
        if not visited[i]:
            perform_DFS(g, i, visited)
            last_v = i

    # If there exist mother vertex (or vetices) in given
    # graph, then v must be one (or one of them)

    # Now check if v is actually a mother vertex (or graph
    # has a mother vertex). We basically check if every vertex
    # is reachable from v or not.

    # Reset all values in visited[] as false and do
    # DFS beginning from v to check if all vertices are
    # reachable from it or not.
    visited = [False]*(g.vertices)
    perform_DFS(g, last_v, visited)
    if any (not i for i in visited): # any() func iterates over a list
        return -1
    else:
        return last_v

# A recursive function to print DFS starting from v
def perform_DFS(g, node, visited):
```

```
    # Mark the current node as visited and print it
    visited[node] = True
    # Recur for all the vertices adjacent to this vertex
    temp = g.array[node].head_node
    while temp:
        if not visited[temp.data]:
            perform_DFS(g, temp.data, visited)
        temp = temp.next_element

if __name__ == "__main__" :
    g = Graph(4)
    g.add_edge(0, 1)
    g.add_edge(1, 2)
    g.add_edge(3, 0)
    g.add_edge(3, 1)
    print(find_mother_vertex(g))
```

This solution is based on **Kosaraju's Strongly Connected Component Algorithm**. Initially, we run the DFS on the whole graph in a loop (line **16**). The DFS ensures that all the nodes in the graph are visited. If the graph is disconnected, the `visited` list will still have some vertices which haven't been set to `True`.

The theory is that the last vertex visited in the recursive DFS will be the mother vertex. This is because, at the last vertex, all slots in `visited` would be `True` (DFS only stops when all nodes are visited). Hence, we keep track of this last vertex using `last_v`.

Then, we reset the `visited` list and run the DFS only on `last_v`. If it visits all nodes, it is a mother vertex. Otherwise, a mother vertex does not exist. The only limitation in this algorithm is that it can detect one mother vertex, even if others exist.

Time complexity#

The DFS of the whole graph works in O(V + E). If a mother ve ex he second DFS takes O(V + E) as well. Therefore, the complete time complexity for this algorithm is *O(V + E)*.

Back

Challenge 4: Find a "Mother Vertex" in...

Next

Challenge 5: Count Number of Edges i...

Mark as Completed

Report an Issue