Page :
Date : / /

Aman Kumar Pandey
RA1911003010685
Artificial Intelligence Lab
Lab - 4

Aim → Implementation and Analysis of DFS and BFS for an application

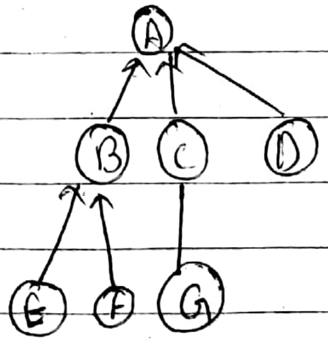
(1) Implementation and Analysis of BFS for Topological sort.

Problem Formulation

Given, a graph with n vertices, display the topological sort of the given graph using Breadth First search (BFS). If the graph contain a cycle no topological sort exists, hence display the message that a cycle exist in the graph.

Initial state

For the given directed graph the initial state of the topological sorting order would be an empty array.
toporder = [ ]



Final state

toporder = [E, F, G, D, B, C, A]

Problem solving

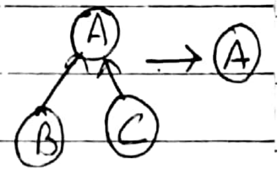
- Since topological ordering consists of those nodes whose indegree (the no. of incoming edges to the vertex) is zero.

  Since E, F, G and D are vertices with zero in degree, they will come first in the topological sort.

- top-order = [E, F, G, D]

- Now the nodes left are A, B and C

- Since after removing E, F, G, D from the graph node B and C are vertices with zero indegree they will get appended to the topological order.



- top-order = [E, F, G, D, B, C]

- Only vertice A is left, therefore the order becomes top-order = [E, F, G, D, B, C, A]

AMAN KUMAR PANDEY

RA1911003010685

ARTIFICIAL INTELLIGENCE LAB

EXPERIMENT NO: 4

IMPLEMENTATION & ANALYSIS OF BFS AND DFS FOR AN APPLICATION

(i) Implementation of BFS for Topological Sort

Algorithm:

Step-1: Start
Step-2: Compute in-degree (number of incoming edges) for each of the vertex present in the DAG and initialize the count of visited nodes as 0.
Step-3: Pick all the vertices with in-degree as 0 and add them into a queue (Enqueue operation)
Step-4: Remove a vertex from the queue (Dequeue operation) and then.

1. Increment count of visited nodes by 1.
2. Decrease in-degree by 1 for all its neighboring nodes.
3. If the in-degree of neighboring nodes is reduced to zero, then add itto the queue.

Step 5: Repeat Step 3 until the queue is empty.
Step 6: If the count of visited nodes is not equal to the number of nodes in the graph, then the topological sort is not possible for the given graph.
Step-7: Stop

Source code:

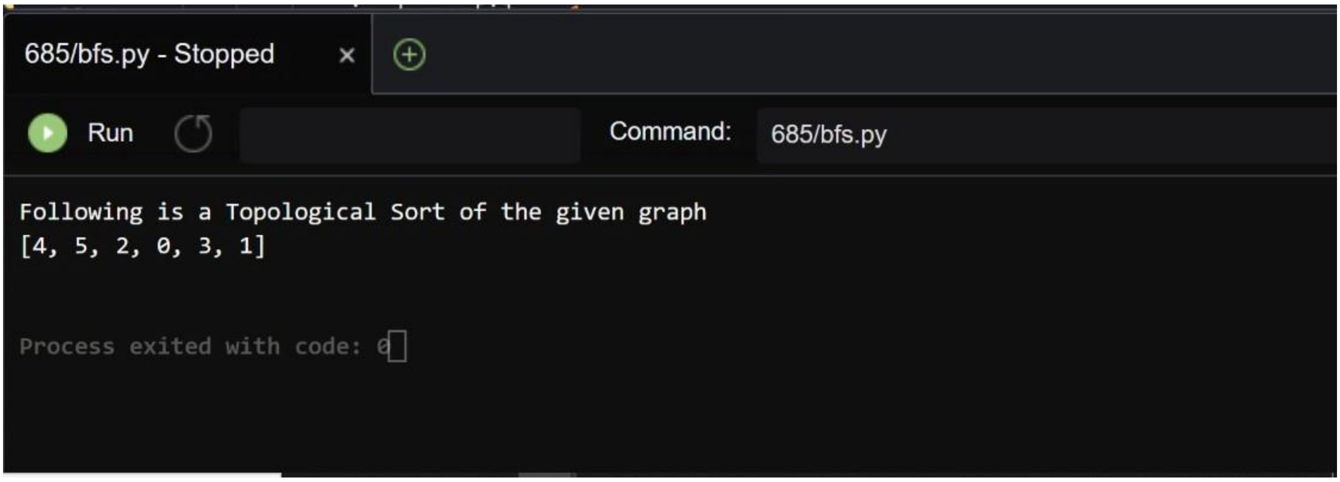
```
from collections import defaultdict
class Graph:
    def__init__(self, vertices):
        self.graph = defaultdict(list)
        self.V = vertices

    def addEdge(self, u, v):
        self.graph[u].append(v)

    def topologicalSort(self):
        in_degree = [0]*(self.V)
        for i in self.graph:
            for j in self.graph[i]:
                in_degree[j] += 1
        queue = []
        for i in range(self.V):
            if in_degree[i] == 0:
                queue.append(i)
        cnt = 0
        top_order = []
        while queue:
            u = queue.pop(0)
            top_order.append(u)
            for i in self.graph[u]:
                in_degree[i] -= 1
                if in_degree[i] == 0:
                    queue.append(i)
            cnt += 1
        if cnt != self.V:
            print ("There exists a cycle in the graph")
        else :
            print (top_order)

g = Graph(6)
g.addEdge(5, 2);
g.addEdge(5, 0);
g.addEdge(4, 0);
g.addEdge(4, 1);
g.addEdge(2, 3);
g.addEdge(3, 1);
print ("Following is a Topological Sort of the given graph")
g.topologicalSort()
```

Output:



Analysis:

- **Time Complexity:** O(V+E).
  The outer for loop will be executed V number of times and the inner for loop will be executed E number of times.

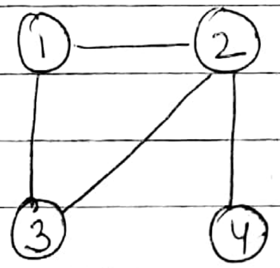
- **Auxiliary Space:** O(V).
  The queue needs to store all the vertices of the graph. So, the space required is O(V)

Page :
Date : / /

ii) Implementation and Analysis of DFS for detecting cycle in an undirected graph.

Problem Formulation

Given a undirected graph with n vertices, check ~~wther~~ whether the graph contains a cycle or not using Depth First Search (DFS). Display a message accordingly.

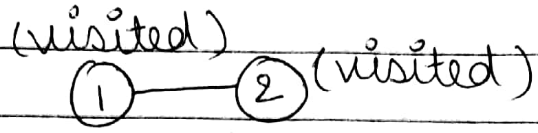


Initial State

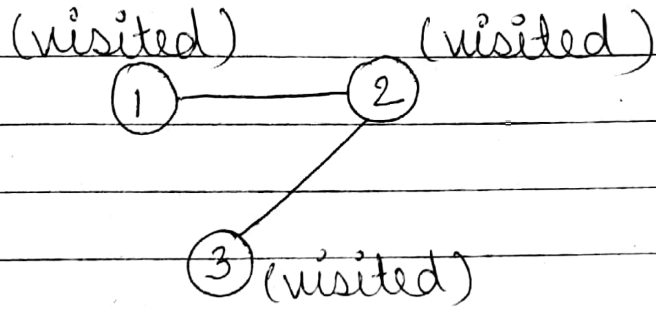
Source node = 1
All the vertices are marked as not visited
visited = [False, False, False, False]

Final State
Since the given graph, contains a cycle.
visited = [True, True, True, False]
cycle exists.

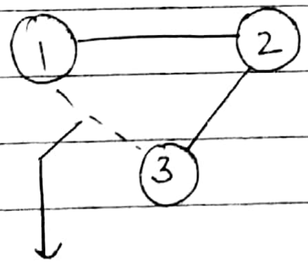
Problem solving
visited = [False, False, False, False]
Start from source node and mark it as visited
1 visited

- visited = [True, False, False, False]
- 

since 1 is 2's parent, hence it is not considered a cycle.

- visited = [True, True, False, False]
- 

visited = [True, True, True, False]



Since on moving from 3 to its adjacent node we encounter vertex 1 which already visited and is not 3's parent. Hence, it implies cycle exists.

- cycle exists.

(ii) Implementation of DFS for Cycle Detection in an undirected graph

Algorithm:

Step-1. Start.
Step-2 Initially mark all the vertices as not visited.
Step-3 Select a vertex and mark it as visited, now move to one of its adjacent vertexes and check if any of its adjacent nodes other than its parent is visited or not.
Step-4 If it is found to be visited then a cycle exists and print a message that "cycle exists" and go to step 5 otherwise repeat step 3 till all the vertex are visited.
Step-5 Stop.

Source code:

```
from collections import defaultdict
 class Graph:
   def__init__(self,vertices):
       self.V= vertices
       self.graph = defaultdict(list)

   def addEdge(self,v,w):
       self.graph[v].append(w)
       self.graph[w].append(v)

   def isCyclicUtil(self,v,visited,parent):
       visited[v]= True
       for i in self.graph[v]:

           if visited[i]==False :
               if(self.isCyclicUtil(i,visited,v)):
                   return True

           elif parent!=i:
               return True
       return  False

   def isCyclic(self):
```

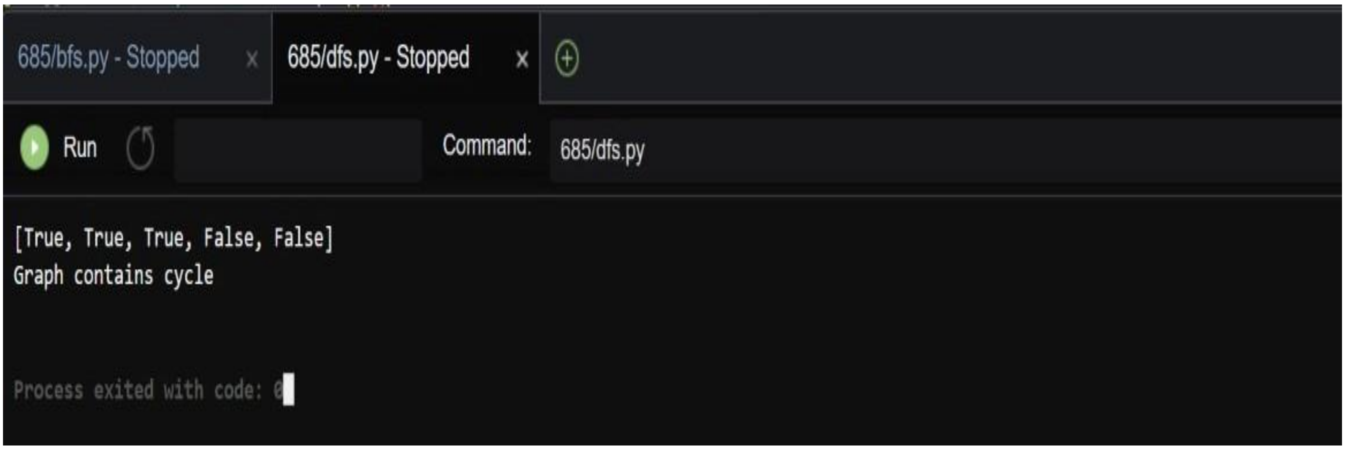
```
        visited =[False]*(self.V)
        for i in range(self.V):
            if visited[i] ==False:
                if(self.isCyclicUtil(i,visited,-1)) == True:
                    print(visited)
                    return True

        return False


g = Graph(5)
g.addEdge(1, 0)
g.addEdge(1, 2)
g.addEdge(2, 0)
g.addEdge(0, 3)
g.addEdge(3, 4)

if g.isCyclic():
    print ("Graph contains cycle")
else :
    print ("Graph does not contain cycle ")
```

Output:



Analysis:

- **Time Complexity:** O(V+E).
  The program does a simple DFS Traversal of the graph which is represented using an adjacency list. So the time complexity is O(V+E).

- **Space Complexity:** O(V).
  To store the visited array O(V) space is required.

Result:

Hence, the implementation of BFS & DFS for an application is donesuccessfully.