Exp 4Implementation and Analysis of DFS and BFS for an application

RA1911031010128-VEER VISWAJEET SWAMI

RA1911031010129-VIJAY RENGARAJ R

RA1911031010130-THINAKAR R

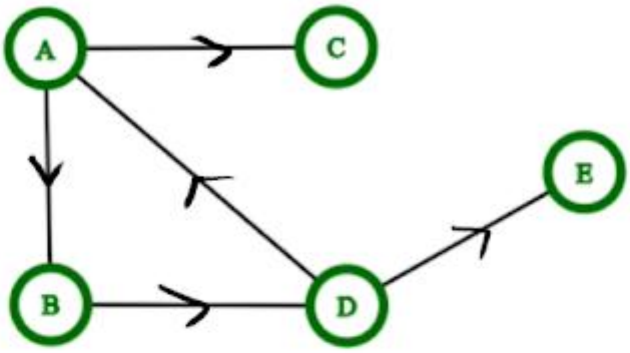
RA1911031010131-NAMARATA MISHRA

RA1911031010138-HUSNA QASIM

Problem chosen: To detect a cycle in a directed graph using DFS and BFS.

Problem Statement:

1)DFS- To detect whether the directed graph contains a cycle or not using DFS.



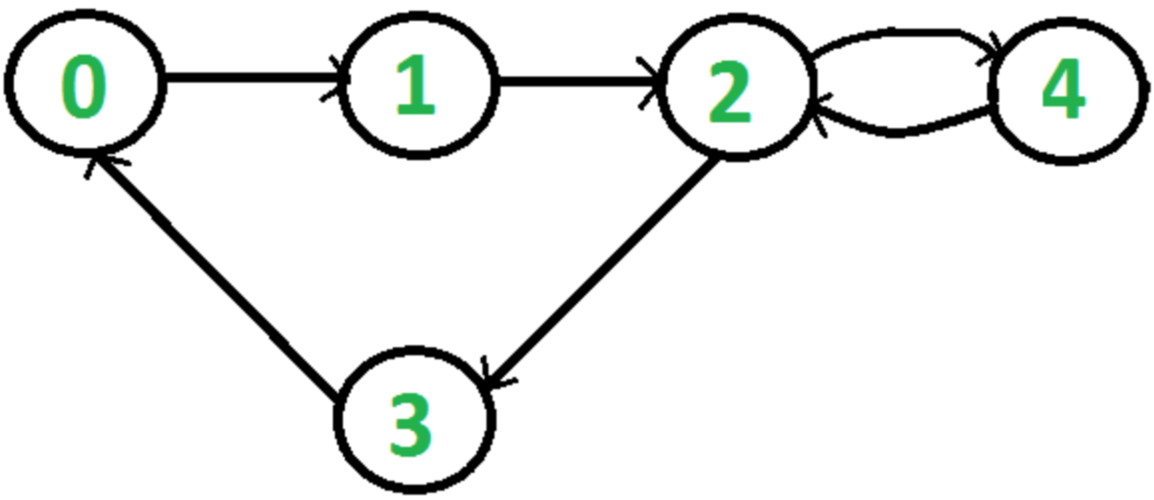
Code and Output:

```
adj_list = {
    "A" : ["C", "B"],
    "B" : ["D"],
    "C" : [],
    "D" : ["A", "E"],
    "E" : []
}

color = {}
parent = {}

for u in adj_list.keys():
    color[u] = 'W'
    parent[u] = None

def dfs(u, color):
    color[u] = 'G'
    print (u)
    for v in adj_list[u]:
        if color[v]=='W':
            cycle = dfs(v, color)
            if cycle:
                return True
        elif color[v]=='G': # cycle is present
            print (u, v)
            return True
    color[u] = 'B'
    return False

is_cyclic = False
for u in adj_list.keys():
    if color[u] == 'W':
        is_cyclic = dfs(u, color)
        if is_cyclic:
            break
print ("Is cyclic " ,is_cyclic)
```

```
In [1]: adj_list = {
            "A" : ["C", "B"],
            "B" : ["D"],
            "C" : [],
            "D" : ["A", "E"],
            "E" : []
        }

        color = {}
        parent = {}

        for u in adj_list.keys():
            color[u] = 'W'
            parent[u] = None

        def dfs(u, color):
            color[u] = 'G'
            print (u)
            for v in adj_list[u]:
                if color[v]=='W':
                    cycle = dfs(v, color)
                    if cycle:
                        return True
                elif color[v]=='G': # cycle is present
                    print (u, v)
                    return True
            color[u] = 'B'
            return False

        is_cyclic = False
        for u in adj_list.keys():
            if color[u] == 'W':
                is_cyclic = dfs(u, color)
                if is_cyclic:
                    break
        print ("Is cyclic " ,is_cyclic)
```

```
A
C
B
D
D A
Is cyclic  True
```

2)BFS- To detect whether the directed graph contains a cycle or not using BFS.



Code and Output:

```
# A Python3 program to check if there is a cycle in
# directed graph using BFS.
import math
import sys
from collections import defaultdict

# Class to represent a graph
class Graph:
  def __init__(self,vertices):
    self.graph=defaultdict(list)
    self.V=vertices # No. of vertices'

  # function to add an edge to graph
  def addEdge(self,u,v):
    self.graph[u].append(v)

# This function returns true if there is a cycle
# in directed graph, else returns false.
def isCycleExist(n,graph):

  # Create a vector to store indegrees of all
  # vertices. Initialize all indegrees as 0.
  in_degree=[0]*n

  # Traverse adjacency lists to fill indegrees of
  # vertices. This step takes O(V+E) time
  for i in range(n):
    for j in graph[i]:
      in_degree[j]+=1

  # Create an queue and enqueue all vertices with
  # indegree 0
  queue=[]
  for i in range(len(in_degree)):
    if in_degree[i]==0:
      queue.append(i)

  # Initialize count of visited vertices
  cnt=0

  # One by one dequeue vertices from queue and enqueue
  # adjacents if indegree of adjacent becomes 0
  while(queue):

    # Extract front of queue (or perform dequeue)
    # and add it to topological order
    nu=queue.pop(0)

    # Iterate through all its neighbouring nodes
    # of dequeued node u and decrease their in-degree
    # by 1
    for v in graph[nu]:
      in_degree[v]-=1
```

```
            # If in-degree becomes zero, add it to queue
            if in_degree[v]==0:
                queue.append(v)
        cnt+=1

    # Check if there was a cycle
    if cnt==n:
        return False
    else:
        return True


# Driver program to test above functions
if __name__=='__main__':

    # Create a graph given in the above diagram
    g=Graph(6)
    g.addEdge(0,1)
    g.addEdge(1,2)
    g.addEdge(2,0)
    g.addEdge(3,4)
    g.addEdge(4,5)

    if isCycleExist(g.V,g.graph):
        print("Yes")
    else:
        print("No")
```

```
In [2]: import math
import sys
from collections import defaultdict

# Class to represent a graph
class Graph:
    def __init__(self,vertices):
        self.graph=defaultdict(list)
        self.V=vertices # No. of vertices'

    # function to add an edge to graph
    def addEdge(self,u,v):
        self.graph[u].append(v)

 # This function returns true if there is a cycle
 # in directed graph, else returns false.
def isCycleExist(n,graph):

    # Create a vector to store indegrees of all
    # vertices. Initialize all indegrees as 0.
    in_degree=[0]*n

    # Traverse adjacency lists to fill indegrees of
    # vertices. This step takes O(V+E) time
    for i in range(n):
        for j in graph[i]:
            in_degree[j]+=1

    # Create an queue and enqueue all vertices with
    # indegree 0
    queue=[]
    for i in range(len(in_degree)):
        if in_degree[i]==0:
            queue.append(i)

    # Initialize count of visited vertices
    cnt=0
```

```
    # One by one dequeue vertices from queue and enqueue
    # adjacents if indegree of adjacent becomes 0
    while(queue):

        # Extract front of queue (or perform dequeue)
        # and add it to topological order
        nu=queue.pop(0)

        # Iterate through all its neighbouring nodes
        # of dequeued node u and decrease their in-degree
        # by 1
        for v in graph[nu]:
            in_degree[v]-=1

            # If in-degree becomes zero, add it to queue
            if in_degree[v]==0:
                queue.append(v)
        cnt+=1

    # Check if there was a cycle
    if cnt==n:
        return False
    else:
        return True


 # Driver program to test above functions
if __name__=='__main__':

    # Create a graph given in the above diagram
    g=Graph(6)
    g.addEdge(0,1)
    g.addEdge(1,2)
    g.addEdge(2,0)
    g.addEdge(3,4)
    g.addEdge(4,5)

    if isCycleExist(g.V,g.graph):
        print("Yes")
    else:
        print("No")
```

```
Yes
```

Result :

The problem statements for both DFS and BFS are solved.