**Question 1🡪 You have given an undirected and connected graph G=(V, E), your task is to determine whether G contains a cycle or not.**

**DFS approach to the question:**

Solution**:**

This is a recursive algorithm in which graph is represented using adjacency list. A node from G[] is considered randomly and marked as done. Now its neighbours (except it’s parent) are also explored similarly and marked as done recursively. When a node is considered, if it is already done then it indicates that there is cycle in the graph as the node is accessed in other than it got accessed previously. Thus the program terminates returning value that indicates presence of cycle.

**Algorithm:**

Let G(V,E) is an undirected graph, represented by adjacency list

🡪done[i] be an element in binary array initialised with 0 representing whether i’th node is done/reached which means it has been traversed or not and then if it is reached , then done[i]=1

🡪neighbour(i) indicates set of nodes to which node i is connected

🡪present is a node that is being accessed to check cycle in graph, initialised with a random node in graph Gr.

🡪p is a node which is assumed as the parent of present node. It is initialised with NULL

Cyclic(G[],done[],present, p)

{

if (done[present]==1)

X again, indicating presence of cycle

{

Print(“cycle is present”);

Return;

}

else

{

done[present]=1

For each node V in neighbour(present) except p

Cyclic(Gr[],done[],V, present)

}

}

**Time Complexity:**

It investigates and checks vertices if there are done. When a vertex is stamped, in the event that it is investigated again it implies that it is diagram contains cycles and program ends. Consequently generally speaking running time in DFS calculation is O(V+E), straight in size of the diagram.

**BFS approach to the question:**

**Solution:** This is fundamentally the same as DFS. Here each hub in the current level is gotten to and checked for the cycle before heading off to the following (level of its youngsters). Here chart is spoken to utilizing nearness list. A hub from G[] is considered arbitrarily and set apart as done. Presently its neighbours (aside from its parent) are driven into line and they are investigated first before the entirety of their kids insertd. At the point when a hub is thought of, on the off chance that it is as of now done, at that point, it demonstrates that there is the cycle in the chart as the hub is gotten to in another way than it got to beforehand. Therefore the program ends returning worth that demonstrates the nearness of cycle.

**Algorithm:**

🡪 Let G(V,E) is an undirected graph, represented by adjacency list

🡪done[i] be an element in binary array initialised with 0 representing whether i’th node is done or not (if it is done, then done[i]=1)

🡪neighbour(i) indicates set of nodes to which node i is connected

🡪 present is a node that is being accessed to check cycle in graph, initialised with a random node in graph Graph.

🡪p is a node which is assumed as the parent of present node. It is initialised with NULL

🡪Let Queue[] be an queue of nodes, initially being empty.

Cyclic(Graph[], done[], present)

{

present= n;//any random node in graph

done[present]=1;

insert (present,Q);

While Q is not empty

{ V=delete(Q)

if (done[V]==1) done X again, indicating presence of cycle

{

Print (“cycle is present”);

Return;

}

else

{

done[V]=1

for each neighbour i of V

insert(neighbour(i), Q)

}

}

Print (“No cycle found”);

}

**Time Complexity:**

Assuming that the graph is connected and represented by adjacency lists.

The operations of enqueuing and dequeuing take O(1) time, and so the total time devoted to queue operations is O(V). Because the procedure scans the adjacency list of each vertex only when the vertex is deleted, it scans each adjacency list atmost once. Since the sum o fthe lengths of all adjacency lists is O(E), the total time spent in scanning adjacency lists is O(E). Thus the total running time of the BFS procedure is O(V+E).

**Question 2🡪 You have given a directed and connected graph G = (V, E), your task is to determine whether G contains a cycle or not.**

**DFS approach :**

**Solution:**

This is a recursive algorithm in which graph is represented using adjacency list (listing nodes which are directed from present node). A node from G[] is considered randomly and marked as done. Now its neighbours (except it’s parent) are also explored similarly and marked as done recursively. When a node is considered, if it is already done then it indicates that there is cycle in the graph as the node is accessed in other path than it was accessed previously as it was accessed previously. Thus the program terminates returning value that indicates presence of cycle. Here it is assumed that the graph is strongly connected. If it is not strongly connected, then in the below DFS algorithm instead of ‘present’ initialised with one value, it should be checked with every node initialised to ‘present’ i.e., algorithm should be checked for every node in Graph as starting node.

**Algorithm:**

🡪 Let G(V, E) is an directed graph, represented by adjacency list

🡪done[i] be an element in binary array initialised with 0 representing whether i’th node is done or not (if it is done, then done[i]=1)

🡪neighbour(i) indicates set of nodes to which edges are directed from i.

🡪 present is a node that is being accessed to check cycle in graph, initialised with a random node in graph G.

Cyclic(G[], done[], present)

{

if (done[present]==1)

{

Print(“cycle is present”);

Return;

}

else

{

done[present]=1

For each node V in neighbour(present)

Cyclic (G[],done[],V)

}

}

**Time Complexity:**

It explores and marks vertices if there are done. Once a vertex is marked, if it is explored again it means that it is graph contains cycles and program terminates. Thus overall running time in DFS algorithm is O(V+E), linear in size of the graph.

**BFS:**

**Solution:**

In BFS algorithm, every node in the present level is accessed and checked for cycle before going to the next level (level of its children). Here graph is represented using adjacency list. A node from G[] is considered randomly and marked as done. Now its neighbours (except it’s parent) are insertd into a queue Q and they are explored first before all their children get explored. When a node is considered, if it is already done then it indicates that there is cycle in the graph as the node is accessed in other path than it got accessed previously. Thus the program terminates returning value that indicates presence of cycle. Here it is assumed that the graph is strongly connected. If it is not strongly connected, then in the below DFS algorithm instead of ‘present’ initialised with one value, it should be checked with every node initialised to ‘present’ i.e., algorithm should be checked for every node in Graph as starting node.

**Algorithm:**

// Let G(V,E) is an undirected graph, represented by adjacency list

//done[i] be an element in binary array initialised with 0 representing whether i’th node is done or not (if it is done, then done[i]=1)

//neighbour(i) indicates set of nodes to which node i is connected

// present is a node that is being accessed to check cycle in graph, initialised with a random node in graph G.

//prev is a node which is assumed as the parent of present node. It is initialised with NULL

//Let Q[] be an queue of nodes, initially being empty.

Cyclic(G[], done[], present)

{

present= n;

done[present]=1;

insert (present,Q);

While Q is not empty

{ V=delete(Q)

if (done[V]==1)

{

Print (“cycle is present”);

Return;

}

else

{

done[V]=1

for each neighbour i of V

insert(neighbour(i), Q)

}

}

Print(“No cycle found!!!”);

}

**Time Complexity:**

Accepting that the diagram is associated and spoken to by contiguousness records.

The activities of enqueuing and dequeuing take O(1) time, thus the complete time dedicated to line tasks is O(V). Since the strategy examines the nearness rundown of every vertex just when the vertex is deleted, it filters every contiguousness list atmost once. Since the whole o fthe lengths of all contiguousness records is O(E), the absolute time spent in filtering nearness records is O(E). Along these lines the absolute running time of the BFS methodology is O(V+E).

**Result🡪**

From the above discussions, it can be stated that the time complexity of detecting cycles in both directed and undirected graphs is same i.e., O(V+E). The optimal running time is found in both DFS and BFS.

One more case for question 1

An undirected graph is acyclic (i.e., a forest) if and only if a DFS yields no back edges. • If there’s a back edge, there’s a cycle. • If there’s no back edge, then by Theorem 22.10, there are only tree edges. Hence, the graph is acyclic. Thus, we can run DFS: if we find a back edge, there’s a cycle. • Time: O(V ). (Not O(V + E)!) If we ever see |V | distinct edges, we must have seen a back edge because (by Theorem B.2 on p. 1174) in an acyclic (undirected) forest, |E| ≤ |V | − 1.