Q2) Given an undirected graph G and an edge uv in G. Design an algorithm that runs in O(|E|+|V|) time that decides if there is a cycle that contains uv.

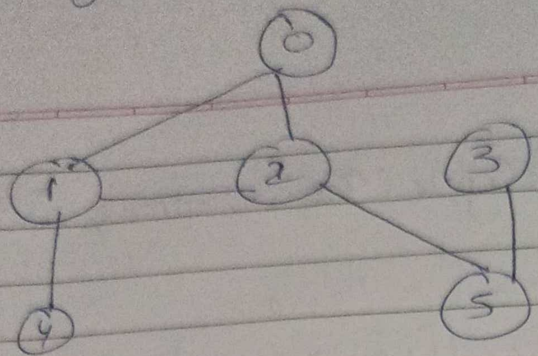
→ ~~A graph G(V,E) has a cycle containing edge e, if there is a path from u to v~~

→ Considering G as graph, and uv as edge that is between any 2 nodes
we have to determine if G has a cycle that contains edge uv. in linear time

→ Algorithm:

1. Create graph with edges & vertices
2. Recursive functn with → current index, visited nodes and previous node
3. Make current node as → visited.
4. Find vertices → not visited & adjacent to current node
   → Call the function again and again for those vertices. If return true → return true.
5. If the neighbor node is not previous & already visited → return true
6. Create a class, that calls the recursive function for all the vertices
   If any functn returns → True → return true
7. Else return false for all vertices.

* Proof with example



RecursiveCycleCheck (0, -1), vis[0] = true

1

RecursiveCycleCheck (1, 0), vis[1] = true

0 — V[0] already visited
* i = parent

2 — RecursiveCycleCheck (2, 1), vis[2] = true

0 — V[0] already visited
* i != parent

∴ Cycle found

Adjacent list :

0 → 1, 2
1 → 0, 2, 4
2 → 0, 1, 5
3 → 5
4 → 1
5 → 2, 3

→ Lets look at the below ~~alo~~ pseudocode to understand how we can code the same

& We will take 2 functions:

1. CheckCycle() → return true if graph contains cycle
2. RecursiveCheckCycle() → recursive function to detect cycle in subgraph

~~Boolean CheckCycle()~~

```
def CheckCycle():
    visited = []
        for i in range(noVertices):
            visited[i] = false

    for u in range(noVertices):
        if u not in visited:
            if (RecursiveCheckCycle(u, visited, -1))
                return true
return false
```

PAGE No.
DATE / /

```
def RecursiveCheckCycle (noVertices, visited[], parent):
    visited[noVertices] = true
    for i in adjacent[noVertices]:
        if i not in visited:
            if RecursiveCheckCycle(i, visited,
                                   noVertices)):
                return True
        elif i != parent
            return true
    return false
```

-> In 1st function (cycleCheck) we initially mark all vertices as not visited and not part of recursion

-> Then, we call the recursive helper function to detect cycle in different DFS trees

-> We do not recur if u is already visited

-> In the 2nd function (RecursiveCheckCycle) we have used visited[] and parent to detect a cycle in subgraph reachable from vertex

-> Initially we mark the current node as visited

-> We recur for all the vertices adjacent to that vertex

-> If an adjacent is not visited, we recur for that adjacent

-> If an adjacent is visited and not the parent of the current vertex, then a cycle is detected.

PAGE No.
DATE / /

Time Complexity: O(V+E)

→ It is a DFS Traversal represented using adjacency list.
∴ Time complexity → O(V+E)

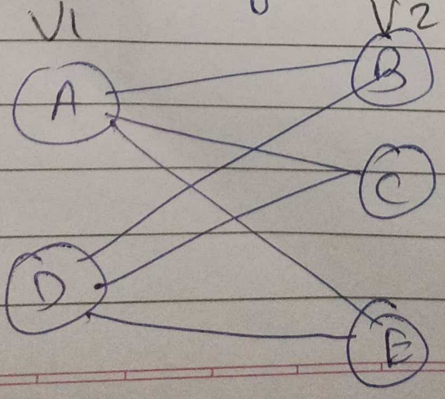
Space complexity: O(V)

Visited array storage will take up O(V) space

Q3] Design an algorithm to tell if an input graph can be colored using 2 colors. (the color constraint is such that no two adjacent vertices have same color).

→ The above algorithm is a bipartite graph problem. A graph is said to be bipartite if:
1. Vertex can be partitioned into 2 disjoint & independents sets. Eg V1 & V2
2. All edges from the graph should have one endpoint vertex from set V1 & another from set V2.

eg



Here we can say that V1 → (A, D)
V2 → (B, C, E)

Both the conditions are satisfied
∴ Bipartite graph

PAGE No.
DATE / /

→ We can use graph coloring & BFS to solve this

Algorithm:

I/p → Graph(G) Vector(V) & Edges(E), Start Vertex(S)
O/p → Can be colored with 2 colors or not.

★ Steps:
1. Assign a red color to the starting vector S
2. Find the neighbors of the starting vertex and assign a blue color.
3. Find neighbor's neighbor and assign a red color.
4. Continue this process, if a neighbor vertex and current vertex has same color then the algo will terminate.

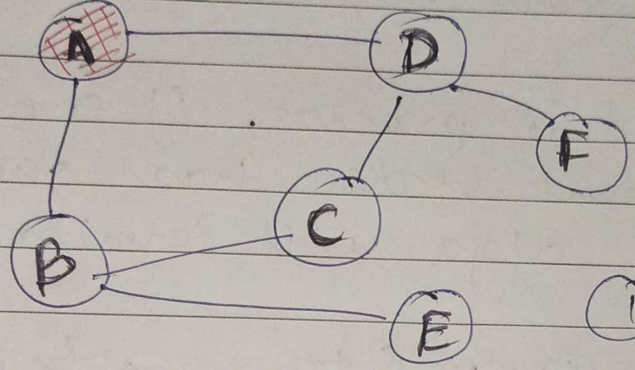
→ We can use Queue Q to save & manage ~~neigh~~ neighbor vertices.

★ Psuedo code:

```
Q = Null
color.StartVertex = Red.
Q. enqueue(StartVertex)
while Q is not empty do
    first_node = Q.dequeue()
    for each node in first_node.adj() do
        if color.node is null do.
            if first_node == red.
                color.node = blue
            else
                color.node = red
            Q.enqueue(node)
```

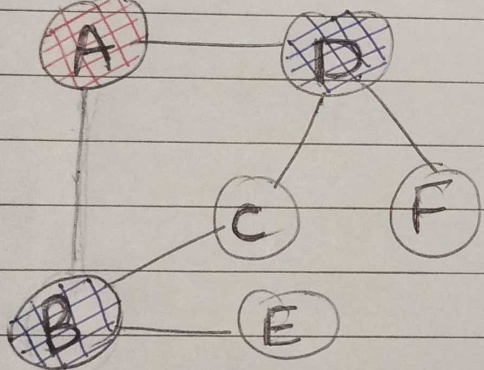
```
            elif color.node == color.first_node then
                 return "Graph cannot be colored"
            end
        end
    end
    return "Graph can be colored"
end
```

☆ Proof of running this algo with eg.

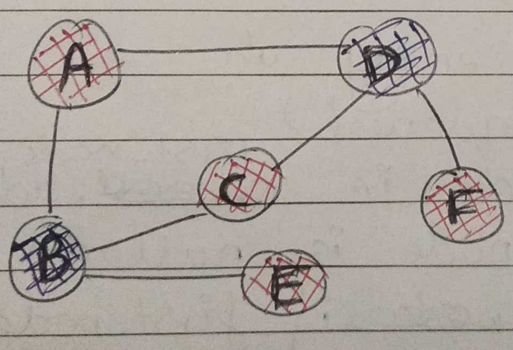


Lets say we have vertex set as (A, B, C, D, E F)

① A → start vertex and red colored.



② Next step is to fill neighbors of A with blue color
Neighbors → B, D



③ → Choosing vertex B, we get vertex C and E as neighbors and we can color them red.

→ we can see, 2 clear partitions
$V_1 = (B, D)$, $V_2 = (A, C, E, F)$
& the algo could color this graph with 2 colors.

④ → Choosing vertex D we get C & F
→ As C is already filled with red color, we color F as red

PAGE No.

DATE / /

* Time Complexity.

→ Since Algo uses BFS, traversing → $O(V+E)$ time.

→ If we use adjacency matrix → it will take $O(V^2)$ time to traverse the vertices in graph

→ If we use adjacency list → $O(V+E)$ time to traverse all vertices & their neighbors

∴ Overall time complexity → $O(V+E)$