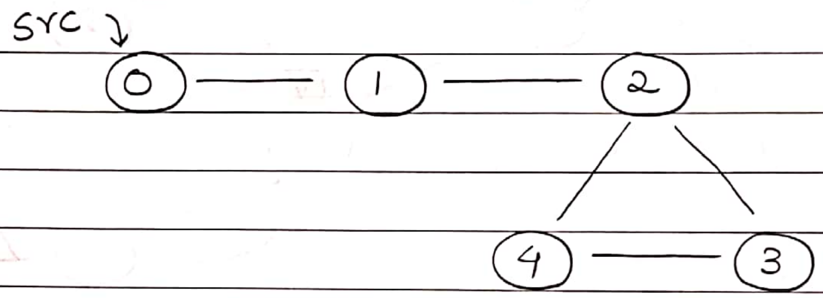
18/06/2023

Cycle Detection in undirected graphs



Here we can see the cycle 2 - 3 - 4.

(i) Using BFS

visited	parent
0 → ~~F~~ T	0 → -1
1 → ~~F~~ T	1 → 0
2 → ~~F~~ T	2 → 1
3 → ~~F~~ T	3 → 2
4 → ~~F~~ T	4 → 2

queue → {0} (mark 0 → T)
Store frontnode, pop it and insert neighbours

1) queue → {1} (Set parent of 1 → 0)
o/p → 0 (mark 1 → T)

2) queue → {2} (Set parent of 2 → 1)
o/p → 0 1 (mark 2 → T)

3) queue → {3, 4} (Set parent of 3 → 2 and 4 → 2)
o/p → 0 1 2 (mark 3 → T and 4 → T)

Now front node is 3, it is now trying to go to 4. 4 is already visited but is not parent of and hence cycle is present.

Note→ Parent will be set only once. If we are trying to set parent of node again, this means cycle is present.

2 conditions → Already visited & is not parent, cycle is present.

Code

```
bool checkCycle (int src, unordered-map <int, bool>
&visited) {

      queue <int> q;
      unordered_map <int, int> parent;
      q.push (src);
      visited [src] = true;
      parent [src] = -1;

      while (!q.empty())
          int frontNode = q.front();
          q.pop();
          for (auto nbr : adjList [frontNode]) {
               if (!visited [nbr]) { // push in queue
                    q.push (nbr);
                    visited [nbr] = true;
                    parent [nbr] = frontNode;
               }
               else { // already visited
```

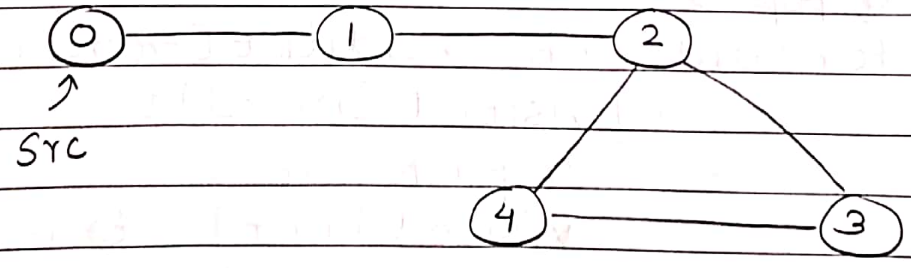
```
            if(nbr != parent[frontNode]){
                  //cycle present
                  return true;
            }
        }
      }
   }
   return false;
}
// In main()
for (int i=0; i<n; i++){
        if(!visited[i]){
              ans = g.checkCycle(i, visited);
              if(ans == true)
                    break;
        }
}
```

ans → true (Cycle present)

ans → false (Cycle not present)

✱ Time complexity = O(V+E)

(ii) Using DFS



```
                                                                       ↑ p=3
dfs(0) → dfs(1) → dfs(2) → dfs(3) → dfs(4)
   ↓         ↓         ↓         ↓        ↓
 p=-1      p=0       p=1       p=2     dfs(2)
```

Here from 4, we are going to 2 which is not parent of 4 and hence cycle is present.

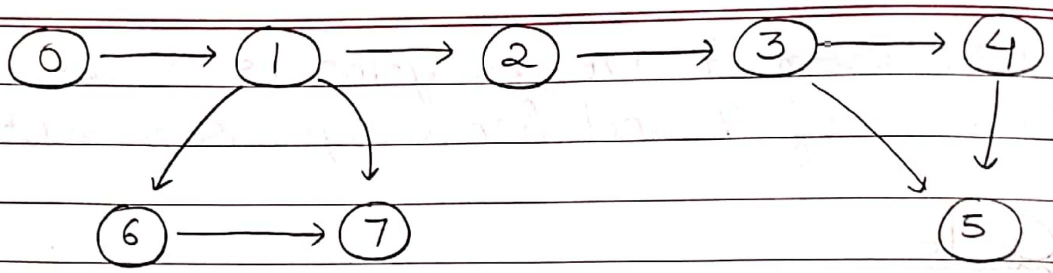
Code

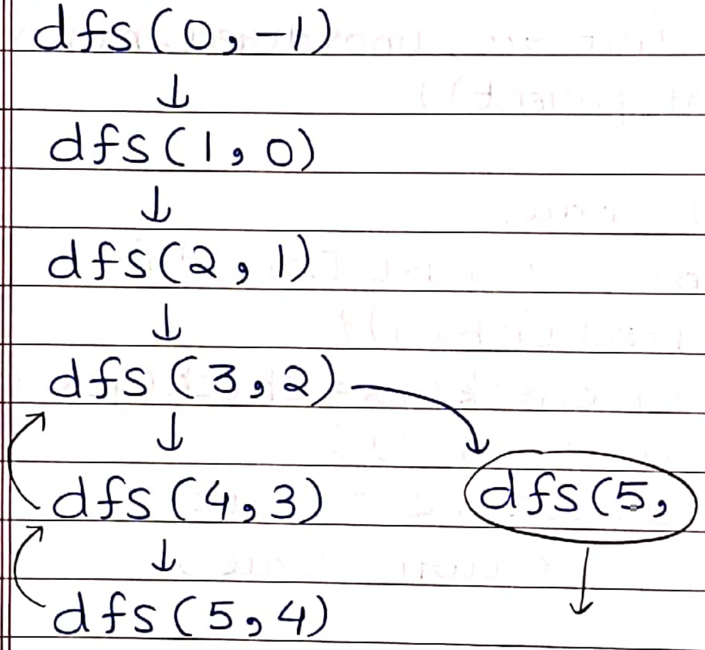
```
bool checkCycle (int src, unordered_map <int,
bool > & visited, int parent) {

    visited [src] = true;
    for (auto nbr : adjList [src]) {
            if (! visited [nbr]) {
                    bool checkAns = checkCycle (src,
                    visited, src);
                    if (checkAns = = true) //cycle present
                            return true;
            }
            else { //already visited
                    //cycle present
                    if (nbr ! = parent) → same condition as BFS.
                            return true;
            }
    }
    return false; //Cycle not present
}
```

Time complexity = O(V+E)

Cycle detection in directed graphs

(i) Using DFS

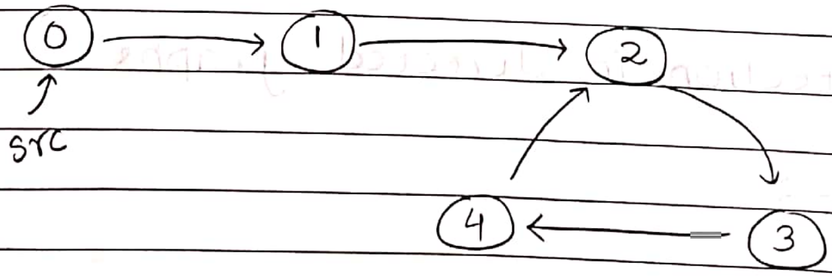




5 is already visited and
5 != parent [3]
5 != 2 (True)
Hence cycle is present but actually cycle is not present.

Hence the previous logic fails. We need to think of some other logic.

New logic



Recursive call stacks

Adjacency list	visited	dfsVisited
0 → 1	0 → ~~F~~ T	0 → ~~F~~ T
1 → 2	1 → ~~F~~ T	1 → ~~F~~ T
2 → 3	2 → ~~F~~ T	2 → ~~F~~ T
3 → 4	3 → ~~F~~ T	3 → ~~F~~ T
4 → 2	4 → ~~F~~ T	4 → ~~F~~ T

dfs(0, -1)
↓
dfs (1, 0)
↓
dfs(2, 1)
↓
dfs(3, 2)
↓
dfs(4, 3)
↓
Trying to go to 2 but dfsVisited has entry for 2 → true and hence cycle is present.

Note → visited → once marked true, can't become false
dfsVisited → once marked true, can become false.

Code

```
bool checkCycle (int src, unordered_map<int,
bool> & visited, unordered_map <int, bool>&
dfsVisited) {

        visited [src] = true;
        dfsVisited [src] = true;
```

```
for (auto nbr : adjList [src]) {
            if (!visited [nbr]) {
                    bool checkAns = checkCycle (nbr,
                    visited, dfsVisited);
                    if (checkAns)
                            return true;
            else {
                    if (dfsVisited [nbr])
                            return true;
            }
        }
        // Backtracking (Here I can do mistake)
        dfs Visited [src] = false ;
        return false ;
}
```

Note→ Rotten oranges + no. of islands are very very important questions. (Solved easily with the help of BFS and DFS)