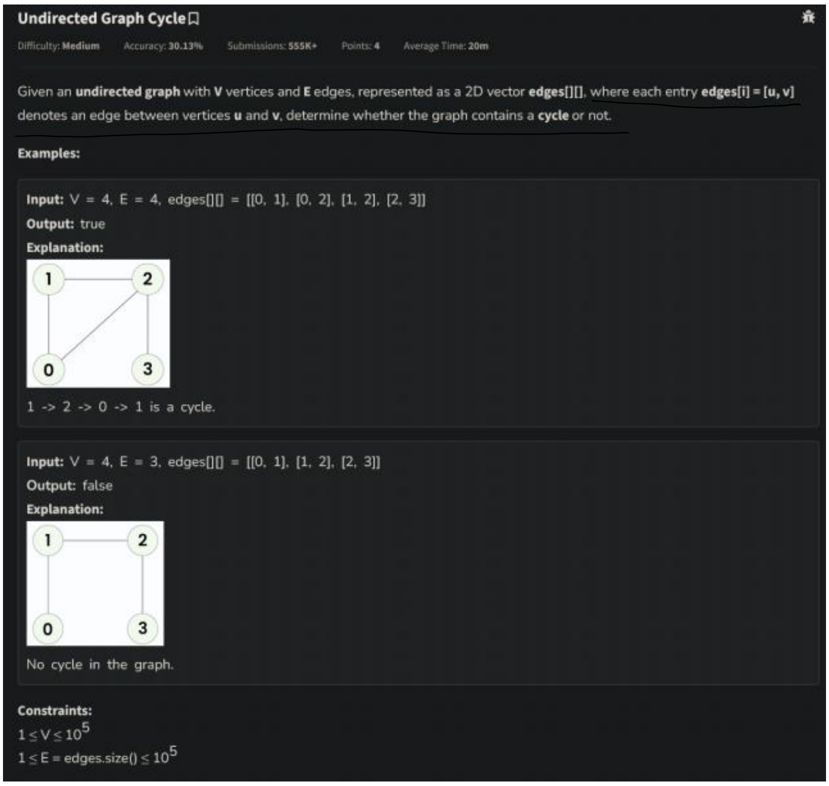
Saturday, April 12, 2025 3:12 PM

Undirected Graph Cycle



Undirected Graph Cycle

Difficulty: **Medium** Accuracy: **30.13%** Submissions: **555K+** Points: **4** Average Time: **20m**

Given an **undirected graph** with **V** vertices and **E** edges, represented as a 2D vector **edges[][]**, where each entry **edges[i] = [u, v]** denotes an edge between vertices **u** and **v**, determine whether the graph contains a **cycle** or not.

Examples:

Input: V = 4, E = 4, edges[][] = [[0, 1], [0, 2], [1, 2], [2, 3]]
Output: true
Explanation:

1 -> 2 -> 0 -> 1 is a cycle.

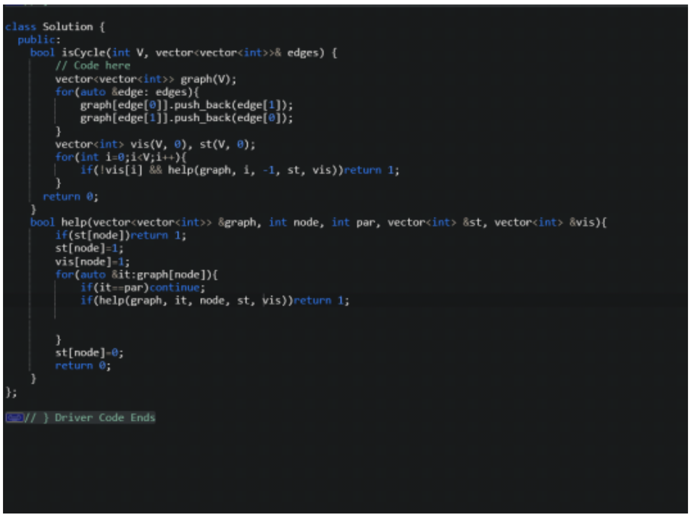
Input: V = 4, E = 3, edges[][] = [[0, 1], [1, 2], [2, 3]]
Output: false
Explanation:

No cycle in the graph.

Constraints:
$1 \leq V \leq 10^5$
$1 \leq E = \text{edges.size()} \leq 10^5$

make use of visited array and a stack array to mark which vertices are in the stack currently.

DFS :-



```
class Solution {
  public:
    bool isCycle(int V, vector<vector<int>>& edges) {
        // Code here
        vector<vector<int>> graph(V);
        for(auto &edge: edges){
            graph[edge[0]].push_back(edge[1]);
            graph[edge[1]].push_back(edge[0]);
        }
        vector<int> vis(V, 0), st(V, 0);
        for(int i=0;i<V;i++){
            if(!vis[i] && help(graph, i, -1, st, vis))return 1;
        }
       return 0;
    }
    bool help(vector<vector<int>> &graph, int node, int par, vector<int> &st, vector<int> &vis){
        if(st[node])return 1;
        st[node]=1;
        vis[node]=1;
        for(auto &it:graph[node]){
            if(it==par)continue;
            if(help(graph, it, node, st, vis))return 1;


        }
        st[node]=0;
        return 0;
    }
};

// } Driver Code Ends
```

T.C. O(V+E)

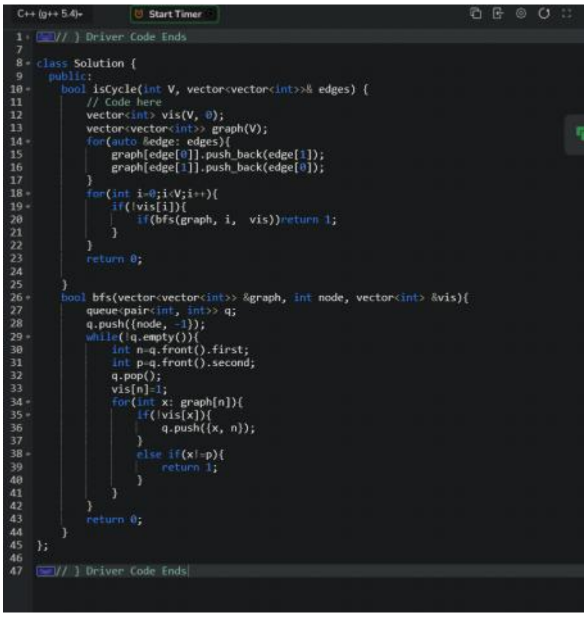
S.C. O(V)

Approach 2 :- BFS :-

The idea is to use BFS to detect a cycle in an undirected graph. we start BFS for all components of the graph & check if a node has been visited earlier, ensuring that we do not consider the parent node of the current node while making this check. If current node while making this check. If we encounter a node that is not the parent, a cycle exists in the graph. Otherwise, we continue BFS by marking the node as visited & inserting it into the queue.

Step by Step Approach:-

1. Initialize a visited array of size n (number of nodes) to false.
2. Iterate through all nodes from 0 to n-1. If a node is not visited, start BFS.
3. Push the node into the queue with its parent set to -1.
4. Perform BFS:
   - Pop a node from the queue.
   - Traverse all its adjacent nodes.
   - If the adjacent node is visited & is not the parent, return true (cycle detected).
   - Otherwise if the adjacent node is not visited, mark it as visited & push it into the queue with the current node as its parent.
5. If no cycle is found after checking all components, return false.



```
// } Driver Code Ends

class Solution {
  public:
    bool isCycle(int V, vector<vector<int>>& edges) {
        // Code here
        vector<int> vis(V, 0);
        vector<vector<int>> graph(V);
        for(auto &edge: edges){
            graph[edge[0]].push_back(edge[1]);
            graph[edge[1]].push_back(edge[0]);
        }
        for(int i=0;i<V;i++){
            if(!vis[i]){
                if(bfs(graph, i,  vis))return 1;
            }
        }
        return 0;

    }
    bool bfs(vector<vector<int>> &graph, int node, vector<int> &vis){
        queue<pair<int, int>> q;
        q.push({node, -1});
        while(!q.empty()){
            int n=q.front().first;
            int p=q.front().second;
            q.pop();
            vis[n]=1;
            for(int x: graph[n]){
                if(!vis[x]){
                    q.push({x, n});
                }
                else if(x!=p){
                    return 1;
                }
            }
        }
        return 0;
    }
};

// } Driver Code Ends
```

Approach 3 :- By DSU :-

```
class dsu{
   private:
   vector <int> rank, parent ;
    public :
    dsu (int n){
        rank.resize (n);
          parent.resize (n);
       for (int i=0; i<n; i++) parent[i]=i; }
      int find( int u) {
          if(u== parent[u]) return u;
          return parent[u] = find (parent(u)); }

       void merge( int u, int v){
         if(rank[u] < rank[v]) swap(u,v);
          parent[v]= u;
         if(rank[u]==rank[v] )rank[u]++; }};

   class Solution{
       public :
          bool isCycle (int V, vector <vector<int>>
                                 & edges){

    dsu  d(V);
for (auto & edge : edges){
   if(d.find (edge[0])==d.find(edge[1]))
      return 1;

   d.merge(edge[0], edge[1]);

}
return 0; }};
```

Time Complexity Analysis of DSU algorithm :-

DSU supports two main operations :-

1. Find(x) - Determines the representative (or parent) of the set containing element x.

2. Union (x, y) - Merges the sets containing elements x & y.

To make DSU efficient, we use 2 optimizations :-

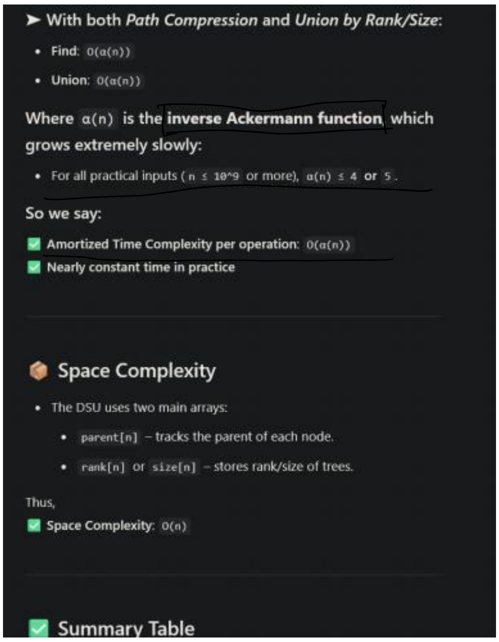
- Path compression (during Find) - makes the tree flat
- Union by Rank or size - keeps the tree shallow during union.

Time Complexity :-

→ without optimizations :-

- Find : O(n) in the worst case
- Union : O(n) in the worst case.

This happens if the trees are skewed (Like a Linked List).



➤ With both *Path Compression* and *Union by Rank/Size*:

- **Find**: `O(α(n))`
- **Union**: `O(α(n))`

Where `α(n)` is the **inverse Ackermann function**, which grows extremely slowly:

- For all practical inputs (`n ≤ 10^9` or more), `α(n) ≤ 4` **or** `5` .

So we say:

✅ **Amortized Time Complexity per operation**: `O(α(n))`
✅ **Nearly constant time in practice**

📦 Space Complexity

- The DSU uses two main arrays:
  - `parent[n]` – tracks the parent of each node.
  - `rank[n]` or `size[n]` – stores rank/size of trees.

Thus,
✅ **Space Complexity**: `O(n)`

✅ Summary Table