**Lecture 37: Graph cycle detection- Union Find and Path Compression Algo, Path Exists in Graph, Invert Binary Tree**

1. [Find if the path Exists in Graph](https://leetcode.com/problems/find-if-path-exists-in-graph/)

Solution Approach: Breadth First Traversal

Time Complexity: O(V+E)

from collections import defaultdict

class Solution:

def validPath(self, n: int, edges: List[List[int]], source: int, destination: int) -> bool:

## Breadth First Traversal

graph = defaultdict(list)

for a, b in edges:

graph[a].append(b)

graph[b].append(a)

## Store all the nodes to be visited in the queue

visited = [False] \* n

visited[source] = True

queue = collections.deque([source])

while queue:

node = queue.popleft()

if node == destination:

return True

for adjacent\_node in graph[node]:

if not visited[adjacent\_node]:

visited[adjacent\_node] = True

queue.append(adjacent\_node)

return False

1. [Invert Binary Tree](https://leetcode.com/problems/invert-binary-tree/)

Solution Approach: Recursion

# Definition for a binary tree node.

# class TreeNode:

# def \_\_init\_\_(self, val=0, left=None, right=None):

# self.val = val

# self.left = left

# self.right = right

class Solution:

def invertTree(self, root: Optional[TreeNode]) -> Optional[TreeNode]:

if root is not None:

rightSubtree = self.invertTree(root.right)

leftSubtree = self.invertTree(root.left)

root.left = rightSubtree

root.right = leftSubtree

return root

1. [Graph cycle detection algorithm: Union by rank and Path Compression algo](https://www.geeksforgeeks.org/union-by-rank-and-path-compression-in-union-find-algorithm/)

class Graph:

def \_\_init\_\_(self, num\_of\_v):

self.num\_of\_v = num\_of\_v

self.edges = defaultdict(list)

def add\_edge(self, u, v):

self.edges[u].append(v)

class Subset:

def \_\_init\_\_(self, parent, rank):

self.parent = parent

self.rank = rank

## Path compression algorithm

def find(subsets, node):

if subsets[node].parent != node:

subsets[node].parent = find(subsets, subsets[node].parent)

return subsets[node].parent

# A function that does the union of two sets

# of u and v(uses union by rank)

def union(subsets, u, v):

# Attach smaller rank tree under root

# of high rank tree(Union by Rank)

if subsets[u].rank > subsets[v].rank:

subsets[v].parent = u

elif subsets[v].rank > subsets[u].rank:

subsets[u].parent = v

# If ranks are the same, then make one as

# root and increment its rank by one

else:

subsets[v].parent = u

subsets[u].rank += 1

# The main function is to check whether a given

# graph contains cycle or not

def isCycle(graph):

# Allocate memory for creating sets

subsets = []

for u in range(graph.num\_of\_v):

subsets.append(Subset(u, 0))

# Iterate through all edges of graph,

# find sets of both vertices of every

# edge, if sets are same, then there

# is cycle in graph.

for u in graph.edges:

u\_rep = find(subsets, u)

for v in graph.edges[u]:

v\_rep = find(subsets, v)

if u\_rep == v\_rep:

return True

else:

union(subsets, u\_rep, v\_rep)

# Driver Code

g = Graph(3)

# add edge 0-1

g.add\_edge(0, 1)

# add edges 1-2

g.add\_edge(1, 2)

# add edge 0-2

g.add\_edge(0, 2)

if isCycle(g):

print('Graph contains cycle')

else:

print('Graph does not contain cycle')