2192. All Ancestors of a Node in a Directed Acyclic Graph Topological Sort Medium Depth-First Search Breadth-First Search Graph

Problem Description In this LeetCode problem, we are tasked with finding the ancestors of each node in a Directed Acyclic Graph (DAG). A DAG is a graph

Leetcode Link

The problem makes two things clear:

with directed edges and no cycles—meaning you can't start at a node, follow a path of edges, and return to that same node.

- Nodes are numbered from 0 to n 1.
- ancestor u of a node v is defined as a node from which v can be reached by traversing the directed edges in one or multiple steps. To solve this, we must perform a search that can navigate through the graph's edges backward—from destination to source—to find

The goal is to return a list where each sublist contains the ancestors of the node at that index, sorted in ascending order. An

The edges are presented in a 2D array with each entry [from, to] signifying a directed edge from node from to node to.

all the nodes that can reach the current node, hence identifying its ancestors. Intuition

The intuition behind the solution is based on the concept of Breadth-First Search (BFS). BFS is a traversal algorithm that starts at

one node and explores all its neighbors before moving on to the neighbors' neighbors, and so on. This way, we can traverse the graph in levels, ensuring that we visit nodes in order of their distance from the starting node.

To apply BFS here, we create a graph representation that allows us to know the descendants of each node. By reversing this logic, we can find the ancestors. For every node, we perform a BFS, storing nodes we encounter as ancestors of the nodes they lead to.

Here's the step-by-step intuition for using BFS to find ancestors:

- 1. Initialize Graph Representation: We need an adjacency list to represent our graph for BFS traversal. This graph lists for each
- node (key) which node it points to (values). It translates the edge pairs into this adjacency list format.
- This method ensures that we systematically check for all the possible ancestors of each node and compile a comprehensive list for
- each one.
- Directed Acyclic Graph (DAG), there are no cycles, and the BFS will indeed terminate for each start node. Here's a detailed explanation of how the provided solution implements this approach:
- 2. Ancestor Lists: A list of lists ans is prepared, with each sublist initialized as empty. This list will hold the ancestors for each node

3. Breadth-First Search (BFS): A helper function bfs(s: int) is defined to perform BFS starting with the node number passed as

s. A deque, a double-ended queue from the collections module, is used for efficient append and pop operations from either

1. Graph Representation: A defaultdict of lists is used to create an adjacency list for the graph. This g will map each node to a list

end. The BFS function implements the following steps:

Solution Approach

left of the queue, then iterates over all nodes j that node i points to, according to g[i]. c. Ancestor Recording: If node j has not yet been visited, it is marked visited, added to the queue, and the start node s is added to ans [j] indicating that s is an ancestor of j.

4. Main Loop: The function bfs(i) is called for every node i in the graph. This ensures that BFS is applied from every possible

5. Returning the Result: The list of lists ans containing the ancestors is returned. Due to the nature of BFS and the problem's

By implementing BFS individually from each node, the algorithm captures all the paths leading to a node in a reverse manner,

constraints, the ancestors will already be sorted, as BFS visits nodes level by level and nodes are processed in ascending order.

b. Traversal: The BFS loop continues until there are no nodes left in the queue. It processes each node i by popping from the

a. Initialization: A queue q is initialized with the start node s, and a vis set is used to keep track of visited nodes.

- therefore identifying all ancestors efficiently. Notably, the use of BFS here exploits the DAG's structure, wherein no backtracking is
- Node 0 points to nodes 1 and 2. Node 2 points to nodes 3 and 4. No other direct connections exist.

2. Ancestor Lists Initialization: Create an empty list of ancestors for each node: 1 ans = [[], [], [], [], []]

Continue with nodes added to the queue. Next up is node 1: - Node 1 has no neighbors, thus no more ancestors to add.

10

12

15

18

19

21

25

1 Start from node 0:

8 Next up is node 2:

13 Start from node 1:

16 Start from node 2:

20 Continue with node 3:

22 23 Continue with node 4: - Same as node 3, ans[4] = [2, 0].

26 Nodes 3 and 4 also have no other children, so BFS concludes for node 2.

 Node 0 has no ancestors (since it points to others but no node points to it). Node 1 is only reached from node 0.

4. Return Result: The final result reflects all the ancestors of each node, capturing that:

Nodes 3 and 4 are reached from both node 2 and (indirectly) node 0.

because BFS always processes nodes in ascending order, and the graph is a DAG.

def breadth_first_search(start_node: int): queue = deque([start_node]) # Queue for BFS visited = {start_node} # Set to keep track of visited nodes while queue: current = queue.popleft()

graph = defaultdict(list)

for parent, child in edges:

for node in range(node_count):

return ancestors

Java Solution

/**

* @param n

breadth_first_search(node)

graph[parent].append(child)

ancestors = [[] for _ in range(node_count)]

Return the list of ancestors for each node

for neighbor in graph[current]:

if neighbor not in visited:

visited.add(neighbor)

queue.append(neighbor)

Initialize a list to store the ancestors for each node

Traverse all nodes and apply BFS to find the ancestors

* Method to find all ancestors of each node in a directed graph.

The number of nodes in the graph.

class Solution:

11

12

13

14

15

16

17

18

20

21

23

24

26

27

28

29

30

32

Node 2 is also only reached from node 0.

- // Create necessary class variables. private int numVertices; // Number of vertices in the graph private List<Integer>[] graph; // Adjacency list representation of graph private List<List<Integer>> ancestors; // List to store ancestors for each node 6
- graph[edge[0]].add(edge[1]); 20 21 ancestors = new ArrayList<>(); 23 for (int i = 0; i < n; ++i) { 24 ancestors.add(new ArrayList<>()); 25 26 27 // Perform a breadth-first search for each node. 28 for (int i = 0; i < n; ++i) { 29 bfs(i); 30 31 return ancestors; 32 33 34 /** 35 * Helper method to perform breadth-first search. 36 37 * @param startVertex The starting vertex for BFS. 38 */ private void bfs(int startVertex) { 39 40 Deque<Integer> queue = new ArrayDeque<>(); // Queue for managing BFS traversal queue.offer(startVertex); 41 42 boolean[] visited = new boolean[numVertices]; // Keep track of visited vertices 43 visited[startVertex] = true; 44 while (!queue.isEmpty()) { 45 46

35 36 37 38 39 40

26

28

29

30

31

32

33

34

6

8

10

11

12

13

14

15

16

17

18

35

37

36 }

C++ Solution

#include <vector>

#include <cstring>

using namespace std;

visited[startNode] = true;

int currentNode = q.front();

// Visit all adjacent nodes

if (!visited[adjNode]) {

q.push(adjNode);

for (int adjNode : graph[currentNode]) {

visited[adjNode] = true;

1 // Function to get all the ancestors for each node in a directed graph

graph[from].push(to); // Add edge to adjacency list

// Initialize an array to hold the ancestors of each node

const breadthFirstSearch = (sourceNode: number) => {

// Create an adjacency list to represent the graph

for (const [from, to] of directedEdges) {

while (queue.length) {

function getAncestors(nodeCount: number, directedEdges: number[][]): number[][] {

const ancestorsList: number[][] = Array.from({ length: nodeCount }, () => []);

const currentNode = queue.shift(); // Get the next node from the queue

const graph: number[][] = Array.from({ length: nodeCount }, () => []);

// Function to perform breadth-first search from a given source node

visited[sourceNode] = true; // Mark the source node as visited

const queue: number[] = [sourceNode]; // Queue for BFS

while (!q.empty()) {

q.pop();

#include <queue>

class Solution {

}; 27 28 29 30

Space Complexity:

return ancestorsList;

Time and Space Complexity The given Python code defines a class Solution with a method getAncestors that calculates the ancestors of each node in a directed acyclic graph (DAG). It performs a breadth-first search (BFS) from every node. Time Complexity: The time complexity of this code is O(V * (V + E)) where V is the number of vertices (n in the code) and E is the number of edges in the graph. Here's the breakdown:

• The graph is represented using an adjacency list (g in the code), which consumes O(E) space. The ans list, which stores the ancestors for each node, will have O(V * A) space complexity where A is the average number of

- however, it is generally bounded by the number of edges E. The BFS uses a queue and a visited set, which in the worst-case can store all vertices; this implies an additional O(V) space. Considering the worst-case scenario for both the adjacency list and the ancestor lists, the space complexity can be bounded by O(V
- the worst conditions and generalize the space complexity as O(V + E) or $O(V^2)$ in the worst case. Note: The exact bounds for the space complexity can be more precisely determined based on the structure of the graph and the distribution of ancestors across the nodes.

- - 2. Ancestor Search via BFS: For each node in the graph, perform a BFS, keeping track of visited nodes to avoid reprocessing. Start BFS at the current node. For each node encountered, mark the starting node as its ancestor.

As this is ancestral tracking and not regular BFS, when a node is visited during traversal, the BFS starting node is recorded

- as an ancestor, rather than the visited node itself. 3. Recording Ancestors: During the BFS, when a new node is visited, add the BFS's starting node to this new node's ancestor list.
- The solution makes use of Breadth-First Search (BFS) to traverse the graph and record the ancestors for each node. Since it is a
- at the index corresponding to the node number.

of nodes to which it points, effectively representing all outgoing edges from every node.

Example Walkthrough

This means:

1 Edges: [[0, 1], [0, 2], [2, 3], [2, 4]]

This graph represents our DAG.

starting node, and all ancestor relationships are explored.

necessary, as no cycles can lead back to the starting node.

Let's consider a small Directed Acyclic Graph with 5 nodes (0 to 4) and the following edges:

1. Graph Representation: We build an adjacency list from the edges:

- Initialize queue with start node (0), and visited set as empty.

1 g = {0: [1, 2], 1: [], 2: [3, 4], 3: [], 4: []}

- No neighbors for node 1, nothing changes.

- Initialize queue with start node (2)

- Following the provided solution approach:
- 3. Breadth-First Search (BFS): We define the BFS helper function and run it for each node. Here's the step-by-step BFS process starting from node 0:

11 Since node 1 had no neighbors, we already completed BFS starting from node 0. Now we continue with the next node.

- Node 0 is visited, add node 0's neighbors to the queue (nodes 1 and 2), and update their ancestor lists: ans[1] = [0], ans[2] =

- Node 2 is visited, add node 2's neighbors to the queue (nodes 3 and 4), and update their ancestor lists: ans[3] = [2], ans[4] =

- Node 2 is visited, no ancestors are added since node 2 is the start node, add node 2's neighbors (nodes 3 and 4) to the queue.

- Node 3 has no neighbors, thus no more ancestors to add. However, we do add node 2 as an ancestor to node 3's ancestors list, so

So the ancestor list for the graph with the given edges is [[], [0], [0], [2, 0], [2, 0]]. Each inode's ancestor list is sorted

After running BFS from all nodes, the ancestor lists are: 1 ans = [[], [0], [0], [2, 0], [2, 0]]

28 Repeat BFS for nodes 3 and 4, but since they don't point to other nodes, there will be no changes.

Python Solution from collections import defaultdict, deque

def getAncestors(self, node_count: int, edges: List[List[int]]) -> List[List[int]]:

Append the start_node as an ancestor of the neighbor

Breadth-first search function to find all ancestors of a given node

ancestors[neighbor].append(start_node)

Create a graph from the edge list, with each node pointing to its successors

- class Solution {
- * @param edges An array of directed edges in the graph. * @return A list of lists containing all ancestors of each node. 13 public List<List<Integer>> getAncestors(int n, int[][] edges) { 14 15 // Initialize the graph representation and the ancestor list. 16 numVertices = n; graph = new List[n]; 17 Arrays.setAll(graph, i -> new ArrayList<>()); 18 for (int[] edge : edges) { 19
- int currentNode = queue.poll(); for (int adjacentNode : graph[currentNode]) { 47 48 // If the adjacent node hasn't been visited yet, // mark it as visited, add it to the queue, 49 50 // and record the current node as its ancestor. 51 if (!visited[adjacentNode]) { 52 visited[adjacentNode] = true; queue.offer(adjacentNode); 54 ancestors.get(adjacentNode).add(startVertex); 55 56 57 58 59 60
- public: vector<vector<int>>> getAncestors(int numNodes, vector<vector<int>>>& edges) { // Graph representation using adjacency lists 10 vector<int> graph[numNodes]; 11 12 // Build the graph from the given edges 13 for (const auto& edge : edges) { 14 graph[edge[0]].push_back(edge[1]); 15 16 17 // Prepare the answer in the form of a vector of vectors 18 vector<vector<int>> ancestors(numNodes); 19 20 // Lambda function for breadth-first search 21 22 auto bfs = [&](int startNode) { 23 queue<int> q; 24 q.push(startNode); 25 vector<bool> visited(numNodes, false);
- }; 41 42 43 // Run BFS for each node to find all ancestors for (int i = 0; i < numNodes; ++i) {</pre> 44 bfs(i); 45 46 47 48 return ancestors; 49 50 }; 51 Typescript Solution

ancestors[adjNode].push_back(startNode); // Add this node as an ancestor

for (const adjacentNode of graph[currentNode!]) { 19 if (!visited[adjacentNode]) { 20 visited[adjacentNode] = true; // Mark as visited 21 22 ancestorsList[adjacentNode].push(sourceNode); // Add the source node to ancestors of the adjacent node 23 queue.push(adjacentNode); // Add adjacent node to the queue for further exploration 24 25 26 // Perform breadth-first search from each node to find all ancestors for (let i = 0; i < nodeCount; ++i) {</pre> 31 breadthFirstSearch(i); 32 33 34 // Return the list of all ancestors for each node

const visited: boolean[] = Array.from({ length: nodeCount }, () => false); // Track visited nodes

- For each of the V vertices, a BFS is executed which can visit all other vertices and traverse through all edges in the worst case. • The BFS itself, for a single source, takes O(V + E) time since each node and edge will be processed at most once.
- The space complexity of the code is O(V + E) for storing the graph and the ancestors:

Therefore, since we execute a BFS for each node, the total time complexity is V times O(V + E), leading to O(V * (V + E)).

- ancestors per node. In the worst case, every node is an ancestor of every other node, which would lead to 0(V^2) space;
- + E). If we account for the worst case with ancestor lists, it tends towards 0(V^2). Therefore, we consider the higher bound given