2192. All Ancestors of a Node in a Directed Acyclic Graph

Medium Depth-First Search Breadth-First Search Graph Topological Sort LeetCode Link

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 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.

The problem makes two things clear:

- Nodes are numbered from 0 to $n - 1$.
- The edges are presented in a 2D array with each entry `[from, to]` signifying a directed edge from node `from` to node `to`.

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

This method ensures that we systematically check for all the possible ancestors of each node and compile a comprehensive list for each one.

Solution Approach

The solution makes use of Breadth-First Search (BFS) to traverse the graph and record the ancestors for each node. Since it is a 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:

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 of nodes to which it points, effectively representing all outgoing edges from every node.
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 at the index corresponding to the node number.
3. **Breadth-First Search (BFS)**: A helper function `bfs(i, int)` is defined to perform BFS starting with the node number passed as i. A `deque`, a double-ended queue from the `collections` module, is used for efficient append and pop operations from either end.

   The BFS function implements the following steps:

   a. **Initialization**: A `deque` q is initialized with the start node i, and a `vis` set is used to keep track of visited nodes.

   b. **Traversal**: The BFS loop continues until there are no nodes left in the queue. It processes each node j by popping from the left of the queue, then iterates over all nodes j that 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 i is added to $ans[j]$ indicating that i 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 starting node, and all ancestor relationships are explored.
5. **Returning the Result**: The list of lists ans containing the ancestors is returned. Due to the nature of BFS and the problem's constraints, the ancestors will already be sorted, as BFS visits nodes level by level and nodes are processed in ascending order.

By implementing BFS individually from each node, the algorithm captures all the paths leading to a node in a reverse manner, therefore identifying all ancestors efficiently. Notably, the use of BFS here exploits the DAG's structure, wherein no backtracking is necessary, as no cycles can lead back to previously visited nodes.

Example Walkthrough

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

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

This means:

- Node 0 points to nodes 1 and 2.
- Node 2 points to nodes 3 and 4. No other direct connections exist.

Following the provided solution approach:

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

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

   This graph represents our DAG.

2. **Ancestor Lists Initialization**: Create an empty list of ancestors for each node:

   ```
   1 ans = [[], [], [], [], []]
   ```

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:

```
1  Start from node 0:
2    - Initialize queue with start node (0), and visited set as empty.
3    - 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] = [0].
4
5  Continue with nodes added to the queue, Next up is node 1:
6    - Node 1 has no neighbors, thus no more ancestors to add.
7
8  Next up is node 2:
9    - 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] = [2].
10
11 Since node 1 had no neighbors, we already completed BFS starting from node 0. Now we continue with the next node.
12
13 Start from node 3:
14   - No neighbors for node 3, nothing changes.
15
16 Start from node 2:
17   - Node 4 is the queue with start node (2)
18   - 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.
19
20 Continue with node 3:
21   - Node 4 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 ans[3] = [2].
22
23 Continue with node 4:
24   - Same as node 3, ans[4] = [2, 0].
25
26 Nodes 3 and 4 also have no other children, so BFS concludes for node 2.
27
28 Repeat BFS for nodes 3 and 4, but since they don't point to other nodes, there will be no changes.
```

After running BFS from all nodes, the ancestor lists are:

```
1 ans = [[], [0], [0], [2, 0], [2, 0]]
```

4. **Return Result**: The final result reflects all the ancestors of each node, capturing that:
   - Node 0 has no ancestors (since it points to others but no node points to it).
   - Node 1 is only reached from node 0.
   - Node 2 is also only reached from node 0.
   - Nodes 3 and 4 are reached from both node 2 (indirectly) node 0.

So the ancestor list for the graph with the edges is `[[], [0], [0], [2, 0], [2, 0]]`. Each index's ancestor list is sorted because BFS always processes nodes in ascending order, and the graph is a DAG.

Python Solution

```
1  from collections import defaultdict, deque
2
3  class Solution:
4      def getAncestors(self, node_count: int, edges: List[List[int]]) -> List[List[int]]:
5          # Breadth-first search function to find all ancestors of a given node
6          def breadth_first_search(start_node: int):
7              queue = deque([start_node])  # Queue for BFS
8              visited = {start_node}  # Set to keep track of visited nodes
9              while queue:
10                 current = queue.popleft()
11                 for neighbor in graph[current]:
12                     if neighbor not in visited:
13                         visited.add(neighbor)
14                         queue.append(neighbor)
15                         # Append the start_node as an ancestor of the neighbor
16                         ancestors[neighbor].append(start_node)
17
18         # Create a graph from the edge list, with each node pointing to its successors
19         graph = defaultdict(list)
20         for parent, child in edges:
21             graph[parent].append(child)
22
23         # Initialize a list to store the ancestors for each node
24         ancestors = [[] for _ in range(node_count)]
25
26         # Traverse all nodes and apply BFS to find the ancestors
27         for node in range(node_count):
28             breadth_first_search(node)
29
30         # Return the list of ancestors for each node
31         return ancestors
32
```

Java Solution

```
1  class Solution {
2      // Create necessary class variables.
3      private List<List<Integer>> graph; // Adjacency list representation of graph
4      private List<List<Integer>> ancestors; // List to store ancestors for each node
5
6      /**
7       * Method to find all ancestors of each node in a directed graph.
8       *
9       * @param n     The number of nodes in the graph.
10      * @param edges 2D array representing the edges of the graph.
11      * @return A list of lists containing all ancestors of each node.
12      */
13     public List<List<Integer>> getAncestors(int n, int[][] edges) {
14         // Initialize the graph representation and the ancestor list.
15         numVertices = n;
16         graph = new List[n];
17         Arrays.setAll(graph, k -> new ArrayList<>());
18         for (int[] edge : edges) {
19             graph[edge[0]].add(edge[1]);
20         }
21
22         ancestors = new ArrayList[n];
23         for (int i = 0; i < n; ++i) {
24             ancestors[i] = 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
32         return ancestors;
33     }
34
35     /**
36      * Helper method to perform breadth-first search.
37      *
38      * @param startVertex The starting vertex for BFS.
39      */
40     private void bfs(int startVertex) {
41         Deque<Integer> queue = new ArrayDeque<>(); // Queue for managing BFS traversal
42         queue.offer(startVertex);
43         boolean[] visited = new boolean[numVertices]; // Keep track of visited vertices
44         visited[startVertex] = true;
45
46         while (!queue.isEmpty()) {
47             for (int currentNode = queue.poll(); // Dequeue the current node
48                  for (int adjacentNode : graph[currentNode]) {
49                      // If the adjacent node hasn't been visited yet,
50                      // mark it as visited, add it to the queue,
51                      // and add the start node to its ancestors.
52                      if (!visited[adjacentNode]) {
53                          visited[adjacentNode] = true;
54                          queue.offer(adjacentNode);
55                          ancestors.get(adjacentNode).add(startVertex);
56                      }
57                  }
58         }
59     }
60 }
```

C++ Solution

```
1  #include <vector>
2  #include <queue>
3  #include <string>
4
5  using namespace std;
6
7  class Solution {
8  public:
9      vector<vector<int>> getAncestors(int numNodes, vector<vector<int>>& edges) {
10         // Graph representation using adjacency lists
11         vector<int> graph[numNodes];
12
13         // Build the graph from the given edges
14         for (const auto& edge : edges) {
15             graph[edge[0]].push_back(edge[1]);
16         }
17
18         // Prepare the answer in the form of a vector of vectors
19         vector<vector<int>> ancestors(numNodes);
20
21         // Lambda function for breadth-first search
22         auto bfs = [&](int startNode) {
23             queue<int> q;
24             q.push(startNode);
25             vector<bool> isVisited(numNodes, false);
26             visited[startNode] = true;
27
28             while (!q.empty()) {
29                 int currentNode = q.front();
30                 q.pop();
31
32                 // Visit all adjacent nodes
33                 for (int adjNode : graph[currentNode]) {
34                     if (!visited[adjNode]) {
35                         visited[adjNode] = true;
36                         ancestors[adjNode].push_back(startNode); // Add this node as an ancestor
37                         q.push(adjNode);
38                     }
39                 }
40             }
41         };
42
43         // Run BFS for each node to find all ancestors
44         for (int i = 0; i < numNodes; ++i) {
45             bfs(i);
46         }
47
48         return ancestors;
49     }
50 };
51
```

Typescript Solution

```
1  // Function to get all the ancestors for each node in a directed graph
2  function getAncestors(nodeCount: number, directedEdges: number[][]): number[][] {
3      // Create an adjacency list to represent the graph
4      const graph: number[][] = Array.from({ length: nodeCount }, () => []);
5      for (const [parent, child] of directedEdges) {
6          graph[parent].push(child); // Add edge to adjacency list
7      }
8
9      // Initialize an array to hold the ancestors of each node
10     const ancestorsList: number[][] = Array.from({ length: nodeCount }, () => []);
11
12     // Function to perform breadth-first search from a given source node
13     const breadthFirstSearch = (sourceNode: number) => {
14         const queue = [sourceNode]; // Queue for BFS
15         const visited: boolean[] = Array.from({ length: nodeCount }, () => false); // Track visited nodes
16         visited[sourceNode] = true; // Mark the source node as visited
17
18         while (queue.length) {
19             const currentNode = queue.shift(); // Get the next node from the queue
20             for (const adjacentNode of graph[currentNode]) {
21                 if (!visited[adjacentNode]) {
22                     visited[adjacentNode] = true; // Mark as visited
23                     ancestorsList[adjacentNode].push(sourceNode); // Add the source node to ancestors of the adjacent node
24                     queue.push(adjacentNode); // Add adjacent node to the queue for further exploration
25                 }
26             }
27         }
28     };
29
30     // Perform breadth-first search from each node to find ancestors
31     for (let i = 0; i < nodeCount; ++i) {
32         breadthFirstSearch(i);
33     }
34
35     // Return the list of all ancestors for each node
36     return ancestorsList;
37 }
```

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:

- 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.

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

Space Complexity:

The space complexity of the code is $O(V + E)$ for storing the graph and the ancestors:

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