797. All Paths From Source to Target

Medium Depth-First Search Breadth-First Search Graph Backtracking

Problem Description

The problem presents us with a directed acyclic graph (DAG) that consists of `n` nodes labeled from `0` to `n - 1`. The goal is to find all the distinct paths that lead from node `0` to node `n - 1` and return the collection of these paths. The definition of the graph is such that for every node `i`, there is a list of nodes, `graph[i]`, that can be reached directly from node `i` through a directed edge. In simple terms, if we can travel from node `i` to node `j`, then `j` would be included in the list that corresponds with `graph[i]`. Since the graph is a DAG, there are no cycles, meaning we won't revisit any node once visited on the same path, which simplifies the traversal process.

Intuition

The key insight to solving this problem lies in understanding that since the graph is acyclic, we can explore it without worrying about getting stuck in a cycle. This allows us to employ depth-first search (DFS) or breadth-first search (BFS) strategies to traverse the graph from the start node (node 0) to the end node (node `n - 1`).

The provided solution uses BFS. The idea behind BFS is to explore the graph level by level, starting from the source node. Here, we initiate a queue (FIFO structure) to keep track of the paths as we discover them. We start by enqueuing the path containing just the source node `[0]`.

For each path taken out of the queue, we look at the last node in the path (current node) and explore all the nodes connected to it as follows:

- If the current node is our destination node (`n - 1`), we've found a complete path from source to destination, so we add it to our list of answers.
- If it's not the destination, we append each neighbor of the current node to a new path and enqueue these new paths back into the queue to be explored later.

This process is repeated until there are no more paths in the queue, meaning we've explored all possible paths from the source to the destination. At the end of this process, the `ans` list contains all unique paths from node `0` to node `n - 1`, and we return it as the final result.

Solution Approach

The given solution employs BFS, a common algorithm used for graph traversal that explores neighbors of a node before moving on to the next level of neighbors. In this approach, a queue is vital, which in Python, can be efficiently implemented using the `collections.deque` allowing for fast appends and pops from both ends.

Here are the steps involved in the solution:

1. Initialize a queue `q` and push the path containing the start node `[0]` onto it.
2. Create a list `ans` to store the answer - all the paths from source to target.
3. While the queue `q` is not empty, repeat the following steps:
   - Pop the first path from the left of the queue (using `popleft()`).
   - Get the last node in the path (current node `u`).
   - If `u` is equal to `n - 1` (target node), then the path is a complete path from source to target. It's then added to the `ans` list.
   - If `u` is not the target, for each neighbor `v` of `u`, create a new path that extends the current path by `v` and enqueue it back into the queue `q` for further exploration.
4. Continue this process until the queue is empty, which means all paths have been explored.
5. Return the `ans` list containing all the successful paths.

By using BFS and a queue, we ensure that each node in a path is only visited once and that all paths are explored systematically. It guarantees that when we reach node `n - 1`, the path we have constructed is a valid path from node `0` to node `n - 1`, and since there are no cycles in a DAG, each path we discover is guaranteed to be a simple path (no repeated nodes).

The use of a path list that is extended and queued at each step avoids mutating any shared state, ensuring that paths discovered in parallel do not interfere with each other. Each discovered path is independent and can be appended to the `ans` list without any additional checks for validity, since the BFS approach inherently takes care of ensuring that a path is not revisited.

In summary, the BFS-based solution is an efficient way to traverse the graph and find all paths from the source to the destination in a DAG.

Example Walkthrough

Given a directed acyclic graph (DAG) defined as `graph = [[1,2], [3], [3], []]`, let's illustrate how the provided BFS solution approach would find all distinct paths from node `0` to node `3`.

- Start by initializing the queue `q` with the path containing just the start node `[0]`. Therefore, `q = [[0]]`.
- The list `ans` to store the answers is initialized as empty: `ans = []`.

Now, we start the BFS process:

1. `q` is `[0]`. We take out `[0]` for processing (dequeue operation).
2. The last node in the path `[0]` is `0`. Since `0` is not the target node (`3`), we look at its neighbors.
3. According to `graph[0]`, the neighbors are `[1, 2]`. So we append each of these to our current path and add these new paths to the queue. Now `q` looks like `[[0, 1], [0, 2]]`.

We repeat these steps until the queue is empty:

4. Process `[0, 1]`. This path ends in `1`, which is not the target.
   - Check neighbors of `1`, which only includes `[3]`.
   - Append `3` to our path, resulting in `[0, 1, 3]`, and add it to the `ans` list since `3` is the target. Queue `q` is now `[[0, 2]]`.
5. Process `[0, 2]`. This path ends in `2`, which is not the target.
   - Check neighbors of `2`, which only includes `[3]`.
   - Append `3` to our path, resulting in `[0, 2, 3]`, and add it to the `ans` list. The queue `q` is now empty.

Now that the queue `q` is empty, we've finished exploring all paths, and the process is complete. The list `ans` contains all complete paths: `ans = [[0, 1, 3], [0, 2, 3]]`.

Each list within `ans` represents a distinct path from node `0` to node `3`. Hence, the paths are:

- Path 1: 0 → 1 → 3
- Path 2: 0 → 2 → 3

These two paths represent all the unique paths through the graph from the start node to the end node, and this is the final answer returned by the BFS approach.

Python Solution

```python
from collections import deque  # Import deque from collections module for efficient queue operations

class Solution:
    def allPathsSourceTarget(self, graph):
        # Determine the number of nodes in the graph
        num_nodes = len(graph)

        # Initialize a queue with the path starting from node 0
        queue = deque([[0]])

        # List to store all possible paths from source to target
        all_paths = []

        # Perform Breadth-First Search
        while queue:
            # Get the first path from the queue
            current_path = queue.popleft()

            # Access the last node in the current_path
            last_node = current_path[-1]

            # If the last node is the target node (last node in graph), append the path to all_paths
            if last_node == num_nodes - 1:
                all_paths.append(current_path)
                continue

            # Explore each neighbor of the last node
            for neighbor in graph[last_node]:
                # Append the neighbor to the current path and add the new path to the queue
                queue.append(current_path + [neighbor])

        # Return the list of all paths from source to target
        return all_paths
```

Java Solution

```java
import java.util.*;

class Solution {
    // Function to find all paths from source (node 0) to target (last node)
    public List<List<Integer>> allPathsSourceTarget(int[][] graph) {
        int n = graph.length; // The number of vertices in the graph
        Queue<List<Integer>> queue = new LinkedList<>(); // Queue to hold the paths to be explored
        queue.offer(Arrays.asList(0)); // Initialize queue with path starting from node 0
        List<List<Integer>> allPaths = new ArrayList<>(); // List to store all the paths from source to target

        // Process paths in the queue
        while (!queue.isEmpty()) {
            List<Integer> path = queue.poll(); // Retrieve and remove the head of the queue
            int lastNode = path.get(path.size() - 1); // Get the last node in the path

            // If the last node is the target, add the path to the result
            if (lastNode == n - 1) {
                allPaths.add(path);
            } else {
                // Explore all the neighbors of the last node
                for (int neighbor : graph[lastNode]) {
                    List<Integer> newPath = new ArrayList<>(path); // Make a copy of the current path
                    newPath.add(neighbor); // Add neighbor to the new path
                    queue.offer(newPath); // Add the new path to the queue
                }
            }
        }

        return allPaths; // Return the list of all paths from source to target
    }
}
```

C++ Solution

```cpp
#include <vector>

using namespace std;

class Solution {
public:
    vector<vector<int>> adjacencyList; // Graph representation as an adjacency list
    vector<vector<int>> allPaths; // To store all paths from source to target

    // Function to find all paths from source to target in a directed graph
    vector<vector<int>> allPathsSourceTarget(vector<vector<int>>& graph) {
        adjacencyList = graph; // Initialize the adjacency list with the graph
        vector<int> currentPath; // Current path being explored
        currentPath.push_back(0); // Start from node 0, as per problem statement
        depthFirstSearch(0, currentPath); // Begin DFS from node 0
        return allPaths; // Return all the computed paths after DFS completion
    }

    // Recursive function to perform depth-first search
    void depthFirstSearch(int nodeIndex, vector<int> currentPath) {
        // Base case: If the current node is the last node in the graph
        if (nodeIndex == adjacencyList.size() - 1) {
            allPaths.push_back(currentPath); // Add current path to all paths
            return; // End recursion
        }

        // Recursive case: Explore all the adjacent nodes
        for (int adjacentNode : adjacencyList[nodeIndex]) {
            currentPath.push_back(adjacentNode); // Add adjacent node to current path
            depthFirstSearch(adjacentNode, currentPath); // Recurse with new node
            currentPath.pop_back(); // Remove the last node to backtrack
        }
    }
};
```

Typescript Solution

```typescript
// Define the function to find all paths from the source (node 0) to the target (last node).
// This function takes a graph represented as an adjacency list and returns an array of paths.
// The graph is an array where graph[i] contains a list of all nodes that node i is connected to.
function allPathsSourceTarget(graph: number[][]): number[][] {
    // Initialize the array to hold all possible paths.
    const paths: number[][] = [];
    // Create a temporary path starting with node 0 (the source).
    const path: number[] = [0];

    // Define the depth-first search function.
    // The 'currentPath' parameter represents the current path being explored.
    const dfs = (currentPath: number[]) => {
        // Get the last node from the current path.
        const currentNode: number = currentPath[currentPath.length - 1];
        // Check if the current node is the target node (last node in the graph).
        if (currentNode === graph.length - 1) {
            // If we've reached the target, add a copy of the current path to the paths array.
            paths.push([...currentPath]);
            return;
        }
        // Iterate over all neighboring nodes connected to the current node.
        for (const nextNode of graph[currentNode]) {
            // Add the neighbor node to the current path.
            currentPath.push(nextNode);
            // Recursively call 'dfs' with the updated path.
            dfs(currentPath);
            // Backtrack: remove the last node from the path to explore other paths.
            currentPath.pop();
        }
    };

    // Start the depth-first search with the initial path.
    dfs(path);
    // Return all the paths found.
    return paths;
}
```

Time and Space Complexity

The provided code is designed to find all paths from the source node (0) to the target node (`n - 1`) in a directed acyclic graph (DAG). Here is an analysis of its time and space complexity:

Time Complexity

The worst-case time complexity is determined by the number of paths and the operations performed on each path. In the worst case, each node except the last can have an edge to every other node, resulting in an exponential number of paths, specifically `O(2^(n-1))`, where `n` is the number of nodes (since each node can be included or not in a path, like a binary decision).

A new list is created for every new path with the operation `path + [v]`, which takes `O(k)` time, where `k` is the length of the current path.

Therefore, the overall worst-case time complexity is `O(2^n * n)`, because there could be `2^n` paths and each path could take up to `n` time to be copied.

Space Complexity

The space complexity is influenced by two factors:

1. The space needed to store all possible paths (`ans`).
2. The additional space needed for the queue (`q`) to store intermediate paths.

In the worst case, all possible paths from the source to the target are stored in `ans`, and each path can be of length `n`, leading to a space complexity of `O(2^n * n)`.

The queue will also store a considerable amount of paths. However, this does not exceed the space complexity for storing all paths since it's essentially a part of the same process.

So, the space complexity of the algorithm is `O(2^n * n)`.