847. Shortest Path Visiting All Nodes

Hard | Bit Manipulation | Breadth-First Search | Graph | Dynamic Programming | Bitmask | Leetcode Link

Problem Description

In this LeetCode problem, we are given an undirected, connected graph with `n` nodes, where nodes are labeled from `0` to `n - 1`. The graph is represented as an adjacency list: `graph[i]` contains all the nodes connected to the node `i` by an edge. Our goal is to determine the shortest path length that visits every node in the graph. We are allowed to start and end at any node, revisit nodes, and traverse any edge multiple times if needed.

The key challenge is to visit all nodes in the most efficient way possible, keeping the path as short as we can manage. It is important to note that the problem isn't asking for the shortest path between two particular nodes, but rather a path that covers all nodes.

Intuition

To solve this problem, we'll use Breadth-First Search (BFS) algorithm paired with state compression to keep track of the nodes visited. The intuition here is that since all edges are equally weighted, BFS is a natural choice to find the shortest path in undirected graphs.

State compression is used here to efficiently represent the set of nodes visited at any point in time. This is crucial because revisiting nodes is allowed, hence the state of our search isn't just the current node, but also the collection of nodes that have been visited so far. By using a bitmask to represent visited nodes (where the `i`-th bit corresponds to whether the `i`-th node has been visited), we can easily update and check which nodes have been seen as we process the BFS queue.

For each node, we start a BFS traversal, where each element in the BFS queue is a pair `(current_node, visited_state)`. We define the initial state as `1 << i`, which means only the node `i` is visited. If the visited state ever becomes `(1 << n) - 1`, it indicates all nodes have been visited, and we have our solution: the length of the path taken to reach this state.

The BFS ensures we find the shortest such path, and the visited set (`vis`) prevents us from processing the same state (node + visited combination) more than once, which is essential to avoid infinite loops and reduce computation.

Solution Approach

The solution leverages the BFS algorithm and bitmasking for state tracking to efficiently determine the shortest path that visits every node in the graph. The approach can be broken down into key components that work synchronously to find the solution:

1. **Breadth-First Search (BFS)**: BFS algorithm is a perfect fit to find the shortest path in a graph with edges of equal weight. It starts traversing from a node and explores all its neighbors before moving onto the neighbors of those neighbors, thus ensuring that the shortest paths to all nodes are identified.

2. **Queue**: The algorithm uses a queue data structure to process nodes in FIFO (first-in-first-out) order, which is the essence of BFS. In Python, this is implemented using the `deque` data structure from the `collections` module for efficient pops from the front of the list.

3. **State Compression with Bitmasking**: To track the nodes that have been visited during the traversal without actually storing all the nodes themselves, state compression through bitmasking is used. For instance, if five nodes have been visited in a graph with `n = 5` nodes, the visited state can be represented as `11111` in binary, which is `31` in decimal. This approach drastically reduces the memory footprint and makes the check for the condition of all nodes being visited (`st == (1 << n) - 1`) straightforward.

4. **Visited Set**: A set named `vis` is used to keep track of all the `(node, state)` pairs that have been visited. This is to avoid repeating searches that have already been done and to ensure we do not go into cycles, which can be a common issue in graph problems.

The implementation steps are as follows:

- Initialize the queue and visited set, and populate them with the first step of the BFS, which includes every node visited on its own first (`(i, 1 << i)`).
- Set `ans` to zero, which will keep track of the number of steps from the start.
- Begin the while loop, which will run until we break out of it when the solution is found.
- Iterate over the queue. Remove (`popleft`) the front of the queue to process the current state. Each state contains a `current_node` and the `visited_state`.
- Check if the visited state signifies that all nodes have been visited by comparing it to `(1 << n) - 1` (which represents a state where all nodes have been visited). If so, this is the shortest path, and we return `ans`.
- If all nodes have not been visited, for each neighbor `j` of the `current_node`, calculate the new state `nst` by setting the corresponding bit (`1 << j`) in the `visited_state`.
- If this new state has not been visited (`if (j, nst) not in vis`), add it to the visited set and queue.
- If the inner loop completes without returning, increment `ans` to indicate that another level of BFS is completed.

Example Walkthrough

Let us consider a small graph represented as an adjacency list for the clarity of our example:

```
graph = [
    [1, 2],    // Node 0 is connected to Node 1 and 2
    [0, 2, 3], // Node 1 is connected to Node 0, 2, and 3
    [0, 1],    // Node 2 is connected to Node 0 and 1
    [1]        // Node 3 is connected to Node 1
]
```

This is an undirected graph with `n = 4` nodes. The goal is to find the shortest path that visits all nodes.

Let's walk through the steps of the BFS algorithm and state compression starting from Node 0:

1. Initialize the BFS queue with our starting nodes and initial state and the visited set `vis`.
    - Queue and `vis` will have the following entries: `{(0, 1), (1, 2), (2, 4), (3, 8)}` where each pair has the format `(current_node, visited_state)`.
    - `1`, `2`, `4`, `8` are the bitmask states for visiting nodes `0`, `1`, `2`, and `3` respectively.
2. `ans` is set to `0`, marking the starting point of the BFS levels.
3. We start our while loop, processing the states in our queue one by one.

Processing `current_node = 0, visited_state = 1`:

- We check each neighbor of node `0`, which is `1` and `2`.
- Update the visited_state to include these neighbors.
- For neighbor `1`, `nst` becomes `1 | (1 << 1)` which is `3` in binary it's `011`.
- For neighbor `2`, `nst` becomes `1 | (1 << 2)` which is `5` in binary it's `101`.
- Since these new states have not been visited, we add them to `vis` and queue: `{(1, 3), (2, 5)}`.

After processing all initial states and their neighbors, we increment `ans` to `1`, which reflects moving to a new level in our BFS search. The queue and `vis` have been updated with the new states.

Queue now looks like:

```
[
  (1, 2), (2, 4), (3, 8), // Remaining initial states
  (1, 3), (2, 5)          // Now states after processing Node 0
] // More states would be added as we continue the BFS
```

This process continues, applying BFS. Each node visits its neighbors, creates new states, adds them to the queue if not already visited, and checks if the visited state equates to `(1 << n) - 1`. If at any point the visited state becomes `1111` (`15` in decimal), this means all nodes have been visited, and the value of `ans` at this point will give us our shortest path length.

Upon finding this condition, the BFS stops, and we return `ans` as the length of the shortest path to visit all nodes.

Python Solution

```python
from collections import deque

class Solution:
    def shortestPathLength(self, graph: List[List[int]]) -> int:
        num_nodes = len(graph)  # Number of nodes in the graph
        queue = deque()
        visited = set()  # Set to store visited (node, state) tuples

        # Initialize the queue and visited set with all individual nodes and their bitmasks
        for node in range(num_nodes):
            state = 1 << node
            queue.append((node, state))
            visited.add((node, state))

        # Initialize the number of steps taken to reach all nodes
        steps = 0

        # Perform a BFS until a path visiting all nodes is found
        while True:
            # Iterate over nodes in the current layer
            for _ in range(len(queue)):
                current_node, state = queue.popleft()
                # Check if the current state has all nodes visited
                if state == (1 << num_nodes) - 1:
                    return steps  # Return the number of steps taken so far

                # Explore neighbors of the current node
                for neighbor in graph[current_node]:
                    new_state = state | (1 << neighbor)
                    # If the new state has not been visited, add it to the queue and set
                    if (neighbor, new_state) not in visited:
                        visited.add((neighbor, new_state))
                        queue.append((neighbor, new_state))

            # After exploring all nodes at the current depth, increment steps
            steps += 1
```

Java Solution

```java
class Solution {
    public int shortestPathLength(int[][] graph) {
        int numNodes = graph.length; // Number of nodes in the graph
        Deque<int[]> queue = new ArrayDeque<>(); // Queue for BFS
        boolean[][] visited = new boolean[numNodes][1 << numNodes]; // Visited states

        // Initialize by adding each node as a starting point in the queue
        // and marking it visited with its own unique state
        for (int i = 0; i < numNodes; ++i) {
            queue.offer(new int[]{i, 1 << i}); // Node index, state as a bit mask
            visited[i][1 << i] = true;
        }

        // BFS traversal
        for (int steps = 0; ; ++steps) { // No terminal condition because the answer is guaranteed to exist
            for (int size = queue.size(); size > 0; --size) { // Process nodes level by level
                int[] pair = queue.poll(); // Get pair (node index, state)
                int node = pair[0], state = pair[1];

                // If the state has all nodes visited (all bits set), return the number of steps
                if (state == (1 << numNodes) - 1) {
                    return steps;
                }

                // Check the neighbors
                for (int neighbor : graph[node]) {
                    // Update the state with the new neighbor; // Set bit for the neighbor
                    // If this state has not been visited before, mark it visited and add to queue
                    if (!visited[neighbor][newState]) {
                        visited[neighbor][newState] = true;
                        queue.offer(new int[] {neighbor, newState});
                    }
                }
            }
        }
    }
}
```

C++ Solution

```cpp
#include <vector>
#include <queue>
#include <cstring>

class Solution {
public:
    // Function to find the shortest path that visits all nodes in an undirected graph.
    int shortestPathLength(std::vector<std::vector<int>>& graph) {
        int nodeCount = graph.size(); // Number of nodes in the graph
        queue<pair<int, int>> nodesQueue; // Queue to maintain the state

        // Visited array to keep track of visited states: node index and visited nodes bitmask
        bool visited[nodeCount][1 << nodeCount];
        memset(visited, false, sizeof(visited)); // Initializing all elements as false

        for (int i = 0; i < nodeCount; ++i) {
            // Initialize the queue with all nodes as starting points
            int bitmask = 1 << i; // Binary representation where only the ith bit is set
            nodesQueue.emplace(i, bitmask);
            visited[i][bitmask] = true; // Set the initial state as visited
        }

        // Loop continually, increasing the path length in each iteration
        for (int pathLength = 0; ; ++pathLength) {
            int levelSize = nodesQueue.size(); // Number of elements at the current level of BFS
            while (levelSize--) { // Loop over all elements in the current BFS level
                auto [currentNode, visitedBitmask] = nodesQueue.front(); // Fetch the front element
                nodesQueue.pop(); // Remove the element from the queue

                // If the state bitmask represents all nodes visited, return the current path length
                if (visitedBitmask == (1 << nodeCount) - 1) {
                    return pathLength;
                }

                // Explore adjacent nodes
                for (int nextNode : graph[currentNode]) {
                    int nextStateBitmask = visitedBitmask | (1 << nextNode); // Update the visited mask
                    // If this state has not been visited
                    if (!visited[nextNode][nextStateBitmask]) {
                        visited[nextNode][nextStateBitmask] = true; // Mark it as visited
                        nodesQueue.emplace(nextNode, nextStateBitmask); // Add to the queue
                    }
                }
            }
        }
    }
};
```

Typescript Solution

```typescript
function shortestPathLength(graph: number[][]): number {
    // The 'n' represents the total number of nodes in the graph.
    const nodeCount = graph.length;

    // The queue to perform BFS. Elements are pairs: [node, state].
    const queue: number[][] = [];

    // Visited states represented by node index and state as a bitmask.
    const visited: boolean[][] = Array.from({ length: nodeCount }, () =>
        new Array(1 << nodeCount).fill(false));

    // Initialize queue and visited with the starting nodes and their respective bitmasks.
    for (let i = 0; i < nodeCount; ++i) {
        const startState = 1 << i; // Initial state for node i (only node i has been visited).
        queue.push([i, startState]);
        visited[i][startState] = true;
    }

    // BFS to find the shortest path length.
    for (let steps = 0; ; ++steps) {
        // Iterate through the current level of the queue.
        for (let size = queue.length; size > 0; --size) {
            const [currentNode, state] = queue.shift()!; // Get next element from the queue.

            // If all nodes have been visited, return the number of steps taken.
            if (state === (1 << nodeCount) - 1) {
                return steps;
            }

            // Check all neighbors of the current node.
            for (const nextNode of graph[currentNode]) {
                const nextState = state | (1 << nextNode); // Mark nextNode as visited in the state.

                // If the state is not yet visited for the nextNode, mark it and enqueue.
                if (!visited[nextNode][nextState]) {
                    visited[nextNode][nextState] = true;
                    queue.push([nextNode, nextState]);
                }
            }
        }
    }
}
```

Time and Space Complexity

Time Complexity

The time complexity of the provided code can be considered as $O(N * 2^N)$ where N is the number of nodes in the graph. Here's the breakdown:

- The algorithm uses Breadth-First Search (BFS) to traverse through every possible state of visited nodes.
- There are 2^N possible states since each node can either be visited or not, marked by bitmasks.
- For each state, we go through all the N nodes and perform constant-time operations (a bitwise OR to update the state and some checks).
- The outer loop iterates until the queue is empty, and since we enqueue each state at most once, it iterates $N * 2^N$ times across all layers of BFS.

Space Complexity

The space complexity is $O(N * 2^N)$, which arises from the following:

- A `vis` set is maintained to keep track of visited states to ensure that each state is processed exactly once. Since there are N nodes and 2^N possible states, the set size can go up to $N * 2^N$.
- A queue `q` is used to perform BFS, this queue will also store at most $N * 2^N$ elements, which are pairs of current node and the state of visited nodes up to that point.