2039. The Time When the Network Becomes Idle

Medium | Breadth-First Search | Graph | Array | Leetcode Link

Problem Description

In this problem, there is a network consisting of `n` servers, each one labeled with a unique number ranging from `0` to `n - 1`. The communication between these servers occurs through edges represented as a 2D array `edges`, where each edge `edges[i] = [u_i, v_i]` is a direct message channel allowing servers `u_i` and `v_i` to exchange an unlimited number of messages in exactly one second.

There is one special server, the master server, labeled as `0`. The rest of the servers are data servers, which need to communicate with the master server. Each data server sends a message that must be processed by the master server. After the processing is done, the master sends a reply back to the data server along the same path the original message took. The time taken for messages to reach the master server is the shortest possible, and the master processes all the messages instantaneously.

A key aspect of the system is the `patience` of each server. The patience is defined by the `patience` array, where `patience[i]` refers to the number of seconds a data server `i` will wait before resending its message if it hasn't received a response. Resending occurs every `patience[i]` seconds until a reply is received.

This problem asks us to find the earliest second when the network becomes idle, meaning no more messages are in transit and no server is awaiting a reply.

Intuition

The intuition behind solving this problem lies in determining the time it takes for the master server to send a reply back to each data server and also accounting for the additional delay caused by the servers' patience in resending messages. The network becomes idle when the last message sent by a data server has been processed by the master server and the reply has been received by the data server.

To calculate the time efficiently, we first represent the network as a undirected graph using the `edges` array, then we use Breadth-First Search (BFS) to calculate the minimum distance from each server to the master server. During the BFS traversal, we also keep track of visited nodes to avoid revisiting nodes, as this would not yield the shortest path.

The key realization is that the time taken for the first message to travel from server `i` to the master server and for the response to come back is twice the distance `d_i` traveled by the message (since the reply follows the same path). If we know how often a server resends its message (`patience[i]`), we can calculate the last time that server will send its message before it becomes idle. This is done by finding the time just before a resend would occur and adding it to the base travel time for the message round-trip, plus 1 second for processing.

By finding the maximum of these latest message times across all servers, we obtain the earliest second when the network becomes idle.

Solution Approach

The implementation of this solution heavily relies on Breadth-First Search (BFS) to traverse the graph constructed from the network of servers and message channels. The BFS traversal allows us to find the shortest path from every data server to the master server. Here's how the algorithm unfolds:

1. We begin by constructing a graph `g` using a `defaultdict` of lists to store the adjacency list of each server. This structure is populated using the given `edges` array.
2. We initialize a queue `q` and enqueue the master server `0`. The queue will be used to perform BFS.
3. A set `vis` is used to keep a record of visited servers to avoid revisiting them as it won't yield the shortest path.
4. The BFS commences as we iterate while the queue is not empty. Each level of BFS represents an increase in distance `d` from the master server. At every level, we process all servers that are `d` distance away.
5. For each server `u` dequeued from the queue, we examine its neighbors `v`. If neighbor `v` has not been visited, we mark it as visited and enqueue it. Then we perform the critical step of calculating the reply time for server `v` from the master server.
6. The reply time for server `v` is based on the server's `patience[v]` level. The calculation `(t - 1) // patience[v] * patience[v] + t + 1` finds the last time server `v` sends a message before it receives a reply and effectively becomes idle. Variable `t` denotes twice the distance from `v` to the master server, considering one trip for the message and one trip for the reply.
7. We maintain a running maximum `ans` of the latest time when any data server sends a message. This accounts for all data servers' resend times and the round trips of their messages.
8. Once BFS traversal is complete, we have the maximum of the latest send times among all servers, which gives us the earliest second the network becomes idle.

In summary, by utilizing BFS for shortest-path calculation and applying the patience and resend logic, the algorithm efficiently determines the earliest time of network idleness.

Example Walkthrough

Let's consider a simple example to illustrate the solution approach:

Given Network:

Servers: `n = 4` (servers `0` to `3`, where `0` is the master server)

Edges (message channels): `edges = [[0, 1], [1, 2], [2, 3]]`

Server patience: `patience = [0, 2, 1, 2]` (not relevant for server `0`)

Graph Representation:

Our graph `g` will look like this after processing the `edges` array:

```
Server 0 -- Server 1 -- Server 2 -- Server 3
```

BFS Initialization:

Queue `q`: initally contains just the master server: `[0]`

Set `vis`: initally empty and will keep track of visited servers: `{}`

Distance `d`: initially zero, as the master server is at a distance of `0` from itself

BFS Traversal:

1. Dequeue server `0`. Since `0` is the master server, there's no need to calculate reply time.
   - Enqueue its only neighbor: Server `1`. Set `vis` to `{1}`.
2. Increment distance `d` to `1`, and process server `1`.
   - Enqueue server `2`. Set `vis` to `{1, 2}`.
   - Calculate reply time for server `1`: (2 * d = 2) for round trip, the last time server `1` would send a message before receiving a reply would be at ((2-1) // 2 * 2 + 2 + 1 = 3). Update running maximum `ans` to `3`.
3. Increment distance `d` to `2`, and process server `2`.
   - Enqueue server `3`. Set `vis` to `{1, 2, 3}`.
   - Calculate reply time for server `2`: (2 * d = 4) for round trip, since `patience[2] = 1`, server `2` would not resend the message before receiving a reply. The reply will be received at (2 * d + 1 = 5). Update running maximum `ans` to `5`.
4. Increment distance `d` to `3`, and process server `3`.
   - Calculate reply time for server `3`: (2 * d = 6) for round trip, the last time server `3` would send a message before receiving a reply would be at ((6-1) // 2 * 2 + 6 + 1 = 7). Update running maximum `ans` to `7`.

Final Result:

Once we've processed all servers, we find that the latest a server (server 3 in this example) sends a message is at second `7`. Therefore, the earliest second when the network becomes idle is `7`.

The network idleness is reached as follows:

- Server `1` sends at `t = 0`, resends at `t = 2`, gets a reply at `t = 3`.
- Server `2` sends at `t = 0`, gets a reply at `t = 5` without resending due to a patience of `1`.
- Server `3` sends at `t = 0`, resends at `t = 2`, gets a reply at `t = 7`.

Thus, by following the solution approach, the BFS traversal helped us determine the shortest path distances, and by applying the patience and resend logic, we could calculate the earliest second when the network becomes idle.

Python Solution

```python
from collections import defaultdict, deque

class Solution:
    def networkBecomesIdle(self, edges: List[List[int]], patience: List[int]) -> int:
        # Create an adjacency list to represent the graph
        graph = defaultdict(list)
        # Initialize the graph with bidirectional edges
        for source, destination in edges:
            graph[source].append(destination)
            graph[destination].append(source)

        # A queue for BFS
        queue = deque([0])
        # Set to keep track of visited nodes
        visited = {0}
        # 'idle_time' keeps track of the maximum time after which the network becomes idle
        idle_time = 0
        # 'distance' represents the current distance from node 0
        distance = 0

        # Perform BFS to calculate the least amount of time
        # when the server becomes idle for each node
        while queue:
            distance += 1
            # Total round trip time for the current level
            trip_time = distance * 2
            # Process all nodes at the current distance/depth
            for _ in range(len(queue)):
                current_node = queue.popleft()
                # Check all adjacent nodes
                for neighbor in graph[current_node]:
                    if neighbor not in visited:
                        # Mark the neighbor as visited and add it to the queue
                        visited.add(neighbor)
                        queue.append(neighbor)
                        # Data packets are sent every 'patience' interval
                        # Find out the last packet time that gets a response
                        last_packet_time = ((trip_time - 1) // patience[neighbor]) * patience[neighbor]
                        # Add trip_time to last_packet_time to get the idle time and then add 1 to account for the final second
                        # when the server finishes processing
                        final_time = last_packet_time + trip_time + 1
                        # Update 'idle_time' to the maximum value we have seen
                        idle_time = max(idle_time, final_time)

        # Return the time after which all nodes have become idle
        return idle_time
```

Java Solution

```java
class Solution {
    public int networkBecomesIdle(int[][] edges, int[] patience) {
        int nodeCount = patience.length;

        // Initialize adjacency list for representing the graph
        List<Integer>[] graph = new List[nodeCount];
        Arrays.setAll(graph, index -> new ArrayList<>());

        // Populate the adjacency list with the given edges
        for (int[] edge : edges) {
            int node1 = edge[0];
            int node2 = edge[1];
            graph[node1].add(node2);
            graph[node2].add(node1);
        }

        // Queue to hold nodes for breadth-first search (BFS)
        Deque<Integer> queue = new ArrayDeque<>();
        queue.offer(0); // Start with node 0 (the master server)

        // Visited array to keep track of visited nodes
        boolean[] visited = new boolean[nodeCount];
        visited[0] = true; // Mark the master server as visited

        int idleTime = 0; // Stores the final result of max idle time
        int distance = 0; // Tracks the distance (in hops) from the master server

        // BFS algorithm to traverse the nodes in the graph
        while (!queue.isEmpty()) {
            distance++;
            // Calculate the total time for a message to go there and back
            int travelTime = distance * 2;
            for (int batchSize = queue.size(); batchSize > 0; batchSize--) {
                int currentNode = queue.poll();
                for (int neighbor : graph[currentNode]) {
                    if (!visited[neighbor]) {
                        visited[neighbor] = true;
                        queue.offer(neighbor);
                        // Calculate message's last transmission time
                        int lastTransmission = (travelTime - 1) / patience[neighbor] * patience[neighbor];
                        // Calculate total idle time for this node until it becomes idle
                        int totalIdleTime = lastTransmission + travelTime + 1;
                        // Update the maximum idle time across all nodes
                        idleTime = Math.max(idleTime, totalIdleTime);
                    }
                }
            }
        }
        return idleTime; // Return the time when the network becomes idle
    }
}
```

C++ Solution

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

class Solution {
public:
    int networkBecomesIdle(vector<vector<int>>& edges, vector<int>& patience) {
        int numServers = patience.size(); // Number of servers in the network
        vector<int> graph[numServers]; // Adjacency list to represent the network

        // Constructing the network graph
        for (auto& edge : edges) {
            int a = edge[0], b = edge[1];
            graph[a].push_back(b);
            graph[b].push_back(a);
        }

        queue<int> bfsQueue; // Queue used for Breadth-First Search (BFS)
        bfsQueue.push(0); // Start from the main server (server 0)
        bool visited[numServers]; // Keep track of visited servers
        memset(visited, false, sizeof(visited)); // Initialize visited array as false
        visited[0] = true; // Mark the main server as visited

        int maxTime = 0; // Variable to store the maximum time when the network becomes idle
        int depth = 0; // Tracks the BFS level or depth

        // BFS to find the shortest path from server 0 to all other servers
        while (!bfsQueue.empty()) {
            depth++;
            int tripTime = depth * 2; // Total round trip time for current depth

            int levelSize = bfsQueue.size(); // Number of nodes in the current level
            for (int i = levelSize; i > 0; --i) {
                int currentServer = bfsQueue.front();
                bfsQueue.pop();

                for (int neighbor : graph[currentServer]) {
                    if (!visited[neighbor]) {
                        visited[neighbor] = true; // Mark the neighbor server as visited
                        bfsQueue.push(neighbor); // Add the neighbor server to the queue

                        // Calculate the last message time for this server
                        int lastMessageTime = ((tripTime - 1) / patience[neighbor]) * patience[neighbor] + tripTime + 1;
                        maxTime = max(maxTime, lastMessageTime); // Update the maximum time
                    }
                }
            }
        }

        return maxTime; // Return the time when the network becomes idle
    }
};
```

Typescript Solution

```typescript
function networkBecomesIdle(edges: number[][], patience: number[]): number {
    const numNodes = patience.length;
    const adjacencyList: number[][] = Array.from({ length: numNodes }, () => []);

    // Construct the graph as an adjacency list
    for (const [node1, node2] of edges) {
        adjacencyList[node1].push(node2);
        adjacencyList[node2].push(node1);
    }

    const visited: boolean[] = Array.from({ length: numNodes }, () => false);
    visited[0] = true; // Starting from the master server node 0

    // Queue for BFS
    let queue: number[] = [0];
    let maxTime = 0;

    // BFS from the master server to calculate the idle time for network
    for (let depth = 1; queue.length > 0; ++depth) {
        const roundTripTime = depth * 2;
        const nextQueue: number[] = [];

        for (const currentNode of queue) {
            for (const neighborNode of adjacencyList[currentNode]) {
                if (!visited[neighborNode]) {
                    visited[neighborNode] = true;
                    nextQueue.push(neighborNode);

                    // Compute the time at which the last data package from this neighbor will arrive
                    const lastDataPackageTime = Math.floor((roundTripTime - 1) / patience[neighborNode]) * patience[neighborNode] +

                    // Update the maxTime if this neighbor's last data package will arrive later than current maxTime
                    maxTime = Math.max(maxTime, lastDataPackageTime);
                }
            }
        }

        queue = nextQueue;
    }

    return maxTime;
}
```

Time and Space Complexity

The time complexity of the code is `O(n + m)`, where `n` is the number of nodes and `m` is the number of edges. The process involves traversing the network graph using a breadth-first search (BFS) algorithm. Each edge is considered exactly once, and each node is processed once during the BFS traversal. Since the number of edges in a network graph can be larger than the number of nodes, the time complexity includes both `n` and `m`.

The space complexity of the code is `O(n + m)` as well since we store the graph as an adjacency list, which requires `O(m)` space. Additionally, the BFS queue can hold at most `O(n)` nodes at any given time, and the `vis` set also requires up to `O(n)` space to track visited nodes.

The reference answer mentioned only `O(n)` complexities, likely under the assumption that the number of edges is proportional to the number of nodes, which is a common characteristic of many network graphs where each node has a limited number of connections. However, for a proper analysis, it is important to consider both the number of nodes and the number of edges.