2039. The Time When the Network Becomes Idle

Medium **Breadth-First Search** Graph Array

### **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

every patience[i] seconds until a reply is received.

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

**Leetcode Link** 

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

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

idle.

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.

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

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

**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.

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.

master server. At every level, we process all servers that are d distance away.

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

second the network becomes idle.

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

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

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

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

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

Here's how the algorithm unfolds:

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] +

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

In summary, by utilizing BFS for shortest-path calculation and applying the patience and resend logic, the algorithm efficiently

t + 1 finds the last time server v sends a message before it receives a reply and effectively becomes idle. Variable t denotes

- determines the earliest time of network idleness. **Example Walkthrough**
- Server patience: patience = [0, 2, 1, 2] (not relevant for server 0) **Graph Representation:**

• Calculate reply time for server 1: (2 \* d = 2) for round trip, the last time server 1 would send a message before receiving a

• Calculate reply time for server 2: (2 \* d = 4) for round trip, since patience[2] = 1, server 2 would not resend the message

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

**BFS Initialization:** 

**BFS Traversal:** 

**Final Result:** 

**Given Network:** 

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

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

• 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}.

3. Increment distance d to 2, and process server 2.

Enqueue server 3. Set vis to {1, 2, 3}.

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.

1. Dequeue server 0. Since 0 is the master server, there's no need to calculate reply time.

reply would be at ((2-1) // 2 \* 2 + 2 + 1 = 3). Update running maximum ans to 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.

The network idleness is reached as follows:

from collections import defaultdict, deque

graph = defaultdict(list)

# A queue for BFS

visited = {0}

idle time = 0

distance = 0

queue = deque([0])

for source, destination in edges:

class Solution:

10

11

12

13

14

15

16

19

20

21

27

28

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

50

52

51 }

C++ Solution

1 #include <vector>

#include <cstring>

class Solution {

public:

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

37

38

39

40

41

43

42 }

2 #include <queue>

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.

Server 1 sends at t = 0, resends at t = 2, gets a reply at t = 3.

Server 3 sends at t = 0, resends at t = 2, gets a reply at t = 7.

# Create an adjacency list to represent the graph

# 'distance' represents the current distance from node 0

# Process all nodes at the current distance/depth

# Perform BFS to calculate the least amount of time

# Initialize the graph with bidirectional edges

graph[source].append(destination)

graph[destination].append(source)

# Set to keep track of visited nodes

for \_ in range(len(queue)):

current\_node = queue.popleft()

for neighbor in graph[current\_node]:

visited.add(neighbor)

queue.append(neighbor)

# Return the time after which all nodes have become idle

if neighbor not in visited:

# Check all adjacent nodes

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

patience and resend logic, we could calculate the earliest second when the network becomes idle.

def networkBecomesIdle(self, edges: List[List[int]], patience: List[int]) -> int:

# 'idle\_time' keeps track of the maximum time after which the network becomes idle

# Mark the neighbor as visited and add it to the queue

# Data packets are sent every 'patience' interval

final\_time = last\_packet\_time + trip\_time + 1

# when the server finishes processing

idle\_time = max(idle\_time, final\_time)

int networkBecomesIdle(vector<vector<int>>& edges, vector<int>& patience) {

queue<int> bfsQueue; // Queue used for Breadth-First Search (BFS)

// BFS to find the shortest path from server 0 to all other servers

bfsQueue.push(0); // Start from the main server (server 0)

bool visited[numServers]; // Keep track of visited servers

visited[0] = true; // Mark the main server as visited

int depth = 0; // Tracks the BFS level or depth

for (int  $i = levelSize; i > 0; --i) {$ 

int currentServer = bfsQueue.front();

if (!visited[neighbor]) {

for (int neighbor : graph[currentServer]) {

// Constructing the network graph

int a = edge[0], b = edge[1];

for (auto& edge : edges) {

graph[a].push\_back(b);

graph[b].push\_back(a);

while (!bfsQueue.empty()) {

bfsQueue.pop();

depth++;

int numServers = patience.size(); // Number of servers in the network

vector<int> graph[numServers]; // Adjacency list to represent the network

memset(visited, false, sizeof(visited)); // Initialize visited array as false

int tripTime = depth \* 2; // Total round trip time for current depth

int levelSize = bfsQueue.size(); // Number of nodes in the current level

int maxTime = 0; // Variable to store the maximum time when the network becomes idle

# Find out the last packet time that gets a response

# Update 'idle\_time' to the maximum value we have seen

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

**Python Solution** 

Thus, by following the solution approach, the BFS traversal helped us determine the shortest path distances, and by applying the

#### # when the server becomes idle for each node while queue: 24 distance += 1 25 # Total round trip time for the current level 26 trip\_time = distance \* 2

```
return idle_time
46
47
Java Solution
    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];
  6
             Arrays.setAll(graph, index -> new ArrayList<>());
  8
             // Populate the adjacency list with the given edges
  9
 10
             for (int[] edge : edges) {
 11
                 int node1 = edge[0];
 12
                 int node2 = edge[1];
 13
                 graph[node1].add(node2);
 14
                 graph[node2].add(node1);
 15
 16
 17
             // Queue to hold nodes for breadth-first search (BFS)
 18
             Deque<Integer> queue = new ArrayDeque<>();
 19
             queue.offer(0); // Start with node 0 (the master server)
 20
 21
             // Visited array to keep track of visited nodes
 22
             boolean[] visited = new boolean[nodeCount];
 23
             visited[0] = true; // Mark the master server as visited
 24
 25
             int idleTime = 0; // Stores the final result of max idle time
 26
             int distance = 0; // Tracks the distance (in hops) from the master server
 27
 28
             // BFS algorithm to traverse the nodes in the graph
 29
             while (!queue.isEmpty()) {
 30
                 distance++;
 31
                 // Calculate the total time for a message to go there and back
 32
                 int travelTime = distance * 2;
 33
                 for (int batchSize = queue.size(); batchSize > 0; batchSize--) {
 34
                     int currentNode = queue.poll();
 35
                     for (int neighbor : graph[currentNode]) {
 36
                         if (!visited[neighbor]) {
 37
                             visited[neighbor] = true;
 38
                             queue.offer(neighbor);
 39
                             // Calculate message's last transmission time
 40
                             int lastTransmission = (travelTime - 1) / patience[neighbor] * patience[neighbor];
 41
                             // Calculate total idle time for this node until it becomes idle
                             int totalIdleTime = lastTransmission + travelTime + 1;
 42
 43
                             // Update the maximum idle time across all nodes
 44
                             idleTime = Math.max(idleTime, totalIdleTime);
 45
 47
 48
 49
             return idleTime; // Return the time when the network becomes idle
```

#### 42 43 44 45

```
39
                             visited[neighbor] = true; // Mark the neighbor server as visited
 40
                             bfsQueue.push(neighbor); // Add the neighbor server to the queue
 41
                             // 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
 46
 47
 48
 49
 50
             return maxTime; // Return the time when the network becomes idle
 51
 52 };
 53
Typescript Solution
     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);
 10
 11
         const visited: boolean[] = Array.from({ length: numNodes }, () => false);
 12
         visited[0] = true; // Starting from the master server node 0
 13
 14
         // Queue for BFS
 15
         let queue: number[] = [0];
 16
         let maxTime = 0;
 17
 18
         // BFS from the master server to calculate the idle time for network
 19
         for (let depth = 1; queue.length > 0; ++depth) {
 20
             const roundTripTime = depth * 2;
 21
             const nextQueue: number[] = [];
 22
 23
             for (const currentNode of queue) {
 24
                 for (const neighborNode of adjacencyList[currentNode]) {
 25
                     if (!visited[neighborNode]) {
 26
                         visited[neighborNode] = true;
 27
                         nextQueue.push(neighborNode);
 28
 29
                         // Compute the time at which the last data package from this neighbor will arrive
 30
 31
 32
                         // Update the maxTime if this neighbor's last data package will arrive later than current maxTime
 33
                         maxTime = Math.max(maxTime, lastDataPackageTime);
 34
 35
 36
```

# Time and Space Complexity

queue = nextQueue;

return maxTime;

const lastDataPackageTime = Math.floor((roundTripTime - 1) / patience[neighborNode]) \* patience[neighborNode] +

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 0(n + m) as well since we store the graph as an adjacency list, which requires 0(m) space. Additionally, the BFS queue can hold at most 0(n) nodes at any given time, and the vis set also requires up to 0(n) space to track visited nodes.

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

The reference answer mentioned only 0(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.