743. Network Delay Time

Depth-First Search

Leetcode Link

Problem Description

The problem presents a directed graph represented by nodes labeled from 1 to n and directed edges with associated travel times.

Graph

Breadth-First Search

The edges are depicted as (u, v, w) tuples, where u is the source node, v is the target node, and w is the time it takes for a signal to travel from the source to the target. The objective is to determine the minimum time required for a signal sent from a given node k to reach all other nodes in the network. If it's not feasible for the signal to reach all nodes, the function should return -1. The challenge lies in finding the shortest path to all nodes from the given starting node to simulate the spread of the signal across the network.

Shortest Path

Heap (Priority Queue)

Intuition

algorithm works:

Medium

1. We initialize a distance array dist to keep track of the shortest distance from the start node to every other node, setting all distances to infinity (INF) except the distance to the start node itself, which is set to 0. 2. We use a minimum heap q to keep track of nodes to visit, prioritized by their current known distance from the start node.

The problem is a classic example of the shortest paths problem, which can be efficiently solved using Dijkstra's algorithm. The

intuition behind the solution is to find the shortest path from the start node k to every other node in the graph. Here's how the

- 3. We repeatedly extract the node u with the smallest known distance from the heap and then relax all edges (u, v, w) going out from u. Relaxing an edge means updating the distance to the target node v if the sum of the current node's distance dist[u] and the edge weight w is less than the known distance dist[v].
- 4. If we find a shorter path to v, we update dist[v] with the new distance and add v to the heap for further consideration. 5. After exploring all reachable nodes, we take the maximum value from the dist array, which will represent the minimum time for
- the last node to receive the signal. 6. If the maximum value is still infinity, which means some nodes are unreachable, we return -1. Otherwise, we return the maximum value as the result.

By using Dijkstra's algorithm, we ensure that we are always exploring the closest node, which allows us to minimize the time for a

Solution Approach

directed graph with non-negative edge weights. The following explains the key portions of the implementation: 1. Graph Representation: The graph g is stored in an adjacency list format using defaultdict(list), where each entry g[u]

contains a list of tuples (v, w) representing an edge from node u to node v with a weight of w.

The given solution leverages Dijkstra's algorithm to compute the shortest paths from a single source node to all other nodes in a

2. Distance Initialization: dist is an array that stores the minimum distance from the start node k to each node. It is initialized to a

signal to propagate to all other nodes in the network.

high value INF = 0x3F3F for all nodes except the start node, which is set to 0, indicating that the distance from the start node to itself is zero. 3. Priority Queue: A priority queue q is implemented using a min-heap via the heapq module. Initially, a tuple (0, k - 1) is pushed

- onto the heap, representing the start node with a distance of 0. 4. Dijkstra's Algorithm: While there are still nodes to be processed in the priority queue (while q:), the node u with the smallest
- if the current known distance to v through u is shorter than any previously known distance (if dist[v] > dist[u] + w:). 5. Relaxation: When a shorter path to a node v is discovered, the distance dist[v] is updated to the new shorter distance dist[u]

+ w, and the new distance along with the node v is pushed onto the heap (heappush(q, (dist[v], v))).

value of ans, which is the minimum time for all nodes to receive the signal.

1. Graph Representation: We initialize an adjacency list g with the directed edges:

from the start node to itself is zero, and all other distances are set to INF.

known distance is popped from the heap. The code then iterates over all neighboring nodes v of u (for v, w in g[u]:), checking

max(dist)). This distance is the minimum time required for the farthest reachable node to receive the signal from node k. 7. Unreachable Nodes: If any node remains at the initialized INF distance, it means that node is unreachable from the start node k.

Therefore, the function will return -1 (return -1 if ans == INF else ans). If all nodes are reachable, the function returns the

6. Result Computation: After all nodes have been processed, the algorithm finds the maximum distance in the dist array (ans =

- With this implementation, Dijkstra's algorithm efficiently finds the shortest paths from the source to all nodes, enabling us to determine the minimum time for network signal propagation. It is important to note that the use of a priority queue optimized with a min-heap is essential for the algorithm's efficiency, as it ensures that the node with the smallest distance is always processed next.
- Let's walk through a small example to illustrate how the solution approach is applied. Consider a directed graph with 4 nodes, labeled from 1 to 4, and let's say we want to find the minimum time for a signal to reach all nodes from node 1. The graph has the following edges with their respective travel times: (1, 2, 4), (1, 3, 2), (2, 4, 1), and (3, 4, 3). Represented visually, the graph looks like this:

We will use the Dijkstra's algorithm to find the shortest path from node 1 to all other nodes. Here's the step-by-step process:

2. Distance Initialization: We create an array dist where dist[0] corresponds to node 1. Set dist[0] to 0 because the distance

3. Priority Queue: We start with a priority queue q and add the start node (0, 0) to it, indicating the node (1-1=0) has a distance of

Each entry represents an edge with the first element as the destination node and the second element as the travel time.

1 q = [(0, 0)]

distance.

3 g[3] = [(4, 3)]

1 g[1] = [(2, 4), (3, 2)] 2 g[2] = [(4, 1)]

1 dist = [0, INF, INF, INF]

Example Walkthrough

1 -> 2 (4)

 $6 \ 3 \rightarrow 4 \ (3)$

4 (1)

3 2 v

4. Dijkstra's Algorithm: Now we enter the main loop, where we visit each node based on a priority determined by the shortest

4, therefore we update dist[1] to 4 and add (4, 1) to the queue. Similarly, for node 3, we find a path with travel time 2. dist[2] is INF, thus we update dist[2] to 2 and add (2, 2) to the queue. Now, our q looks like [(4, 1), (2, 2)].

The next smallest distance in the queue corresponds to node 3; we pop (2, 2) and consider its neighbor node 4. Through node

Now, q has [(4, 1), (5, 3)]. Again, we pop the smallest distance first, which corresponds to node 2. For node 4, we find a

3, the total travel time to node 4 is 2 + 3 = 5. Since dist[3] is INF, we update it to 5 and add (5, 3) to the queue.

better path through node 2 with travel time 4 + 1 = 5. But since dist[3] is already 5, there's no need to update.

Hence, the minimum time needed for a signal to reach all the nodes from node 1 in our example graph is 5 minutes.

Dijkstra's algorithm to find shortest paths from the starting node 'k' to all other nodes.

private static final int MAX_NODES = 110; // assuming the maximum number of nodes is 110

private static final int INFINITY = 0x3f3f3f3f; // represent an infinite distance value

// Define infinity as a large number, larger than any possible path we might calculate.

int networkDelayTime(vector<vector<int>>& times, int n, int k) {

// Fill up the adjacency matrix with provided edge times.

// Vector to keep track of visited nodes, initialized to false.

// Find the unvisited node with the smallest distance (greedy).

if (!visited[j] && (u == -1 || dist[u] > dist[j])) {

dist[v] = std::min(dist[v], dist[u] + graph[u][v]);

// Update the distance to each unvisited neighbor.

vector<vector<int>> graph(n, vector<int>(n, INF));

graph[time[0] - 1][time[1] - 1] = time[2];

// Function to find the time it takes for all nodes to receive the signal from the starting node.

// Initialize the graph's adjacency matrix with infinity (indicating no direct path yet).

// Distance vector to store the shortest time to reach each node, initialized to infinity.

// Calculates the time it takes for all nodes to receive the signal from node k

If a shorter path is found, update the distance and push it to the priority queue.

is. If a node were not reachable, it would still have a value of INF, and we would return -1.

We pop (0, 0) from q since it has the smallest distance, which corresponds to node 1. We then look at all neighbors of node 1.

We go through each node connected to 1. For node 2, we find a path with travel time 4. Since dist[1] is INF, it's greater than 0 +

- 5. Result Computation: After processing all nodes, we have dist as [0, 4, 2, 5]. The maximum value is 5, which is the minimum time required for the signal from node 1 to reach all other nodes. 6. Unreachable Nodes: In this example, all nodes are reachable from the start node, hence our maximum value will be returned as
- def networkDelayTime(self, times: list[list[int]], n: int, k: int) -> int: # A large number to represent infinity. INF = float('inf') # Graph representation using adjacency list where 'g' will hold all the edges.

30 # The time it will take for all nodes to receive the signal. 31 max_delay = max(distances) # If the max_delay is still INF, it means there are nodes that the signal can't reach. 33 return -1 if max_delay == INF else max_delay 34

Java Solution

1 import java.util.Arrays;

class Solution {

Python Solution

class Solution:

10

11

12

13

14

15

16

17

18

20

21

22

23

24

25

26

27

28

29

5

8

9

10

1 from heapq import heappop, heappush

from collections import defaultdict

graph = defaultdict(list)

for source, target, weight in times:

graph[source - 1].append((target - 1, weight))

Min-heap to get the node with the smallest distance.

current_distance, current_node = heappop(min_heap)

Visit all the neighbors of the current node.

new_distance = current_distance + weight

distances[neighbor] = new_distance

heappush(min_heap, (new_distance, neighbor))

if distances[neighbor] > new_distance:

public int networkDelayTime(int[][] times, int n, int k) {

// Initialize the graph with distances set to infinity

for neighbor, weight in graph[current_node]:

Initialize distances to all nodes as infinite except the starting node.

Construct the graph.

distances = [INF] * n

 $min_heap = [(0, k - 1)]$

distances[k-1] = 0

while min_heap:

```
int[][] graph = new int[MAX_NODES][MAX_NODES];
11
            for (int i = 0; i < MAX_NODES; ++i) {</pre>
12
                Arrays.fill(graph[i], INFINITY);
13
14
15
16
            // Fill the graph with input times
            for (int[] edge : times) {
17
                graph[edge[0]][edge[1]] = edge[2];
18
19
20
21
            // Initialize distances from source (node k) to all nodes as infinite
22
            int[] distances = new int[MAX_NODES];
23
            Arrays.fill(distances, INFINITY);
24
            // Distance to the source node itself is always 0
25
            distances[k] = 0;
26
27
            // Keep track of visited nodes
            boolean[] visited = new boolean[MAX_NODES];
28
29
30
            // Perform Dijkstra's algorithm to find shortest paths from node k
31
            for (int i = 0; i < n; ++i) {
32
                int currentNode = -1;
33
                // Find the unvisited node with the smallest distance
34
                for (int j = 1; j \le n; ++j) {
35
                    if (!visited[j] && (currentNode == -1 || distances[currentNode] > distances[j])) {
36
                        currentNode = j;
37
38
39
                // Mark this node as visited
                visited[currentNode] = true;
40
41
42
                // Update distances to neighboring nodes of the current node
                for (int j = 1; j \le n; ++j) {
43
                    distances[j] = Math.min(distances[j], distances[currentNode] + graph[currentNode][j]);
44
45
46
47
            // Find the maximum distance from the source node to all nodes
48
49
            int answer = 0;
            for (int i = 1; i <= n; ++i) {
50
                answer = Math.max(answer, distances[i]);
51
52
53
            // If the maximum distance is INFINITY, not all nodes are reachable
54
55
            return answer == INFINITY ? -1 : answer;
56
57 }
58
```

52 53 54 }; 55

C++ Solution

public:

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

39

40

41

42

43

44

45

46

#include <vector>

class Solution {

#include <algorithm>

static const int INF = 0x3f3f3f3f;

for (const auto& time : times) {

vector<bool> visited(n, false);

// Distance to the starting node is zero.

// Perform relaxation for all nodes.

for (int j = 0; j < n; ++j) {

for (int v = 0; v < n; ++v) {

// Mark the chosen node as visited.

for (int i = 0; i < n; ++i) {

u = j;

visited[u] = true;

vector<int> dist(n, INF);

dist[k-1] = 0;

int u = -1;

```
47
 48
             // Find the maximum distance among all nodes.
             int maxTime = *std::max_element(dist.begin(), dist.end());
 49
 50
 51
            // If the maximum distance is still infinity, not all nodes are reachable.
             return maxTime == INF ? -1 : maxTime;
Typescript Solution
  1 // Set infinity as a large number, larger than any possible path we might calculate.
  2 const INF: number = Number.POSITIVE_INFINITY;
    // Type alias for the times variable, which represents the edge list
    type EdgeList = Array<[number, number, number]>;
    // A method to find the time it will take for all nodes to receive the signal from the starting node.
    function networkDelayTime(times: EdgeList, n: number, k: number): number {
         // Initialize the graph's adjacency matrix with infinity (indicating no direct path yet).
         let graph: number[][] = Array.from({ length: n }, () => Array(n).fill(INF));
 10
 11
         // Fill up the adjacency matrix with the provided edge times.
 12
 13
         times.forEach(([source, target, time]) => {
 14
             graph[source - 1][target - 1] = time;
 15
         });
 16
 17
         // Array to keep track of visited nodes, initialized to false.
         let visited: boolean[] = Array(n).fill(false);
 18
 19
 20
        // Array to store the shortest time to reach each node, initialized to infinity.
 21
         let dist: number[] = Array(n).fill(INF);
 22
 23
         // Distance to the starting node itself is zero.
 24
        dist[k-1] = 0;
 25
 26
        // Perform relaxation for all nodes.
 27
         for (let i = 0; i < n; i++) {
 28
             let u: number = -1;
 29
            // Find the unvisited node with the smallest distance (this is a greedy strategy).
 30
 31
             for (let j = 0; j < n; j++) {
                 if (!visited[j] && (u === -1 || dist[j] < dist[u])) {
 32
 33
 34
 35
 36
 37
            // In the case where disconnected components exist, we would remain with u = -1.
 38
            if (u === -1) continue;
 39
 40
            // Mark the chosen node as visited.
```

Time Complexity The time complexity of the code is determined by the operations performed in the Dijkstra's Algorithm implementation used to

// Example usage:

41

42

43

44

45

46

47

48

49

50

51

52

53

54

55

57

60

heap.

56 }

visited[u] = true;

Time and Space Complexity

compute the shortest paths in the network.

for (let v = 0; v < n; v++) {

let maxTime: number = Math.max(...dist);

return maxTime === INF ? -1 : maxTime;

// networkDelayTime([[2,1,1],[2,3,1],[3,4,1]], 4, 2);

if (!visited[v]) {

// Update the distance to each unvisited neighbor.

dist[v] = Math.min(dist[v], dist[u] + graph[u][v]);

// Find the maximum time among all nodes, which will be the network delay time.

// If the maximum time is still infinity, not all nodes are reachable.

4. Finally, computing the maximum value in the dist array takes O(V) time.

3. The while-loop uses a min-heap (priority queue), which could, in the worst-case scenario, pop and push each vertex once. The heappop operation has O(log V) complexity, and heappush has O(log V) complexity. Since each edge could be relaxed (updated) at most once, if we relax all E edges, this leads to a time complexity of O(E log V) for all heap operations combined.

Adding up all these components, the total time complexity of the function is O(E log V + V), given that initializing the dist array and computing the maximum value are both subsumed by the O(E log V) complexity. **Space Complexity**

1. Building the graph g, which takes O(E) time, where E is the number of edges (or times in this context).

2. Initializing the dist array, which takes O(V) time, where V is the number of vertices (or n in this context).

2. The dist array requires O(V) space. 3. The priority queue (q) at any point stores at most V vertices, hence O(V) space.

1. The graph g stores all edges and requires O(E) space.

Thus, the space complexity of the function is O(E + V), considering both the space for the graph and the auxiliary data structures used.

The space complexity of the code is mainly determined by the storage for the graph representation, the dist array, and the min-