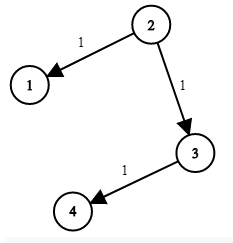
<https://leetcode.com/problems/network-delay-time/>

You are given a network of n nodes, labeled from 1 to n. You are also given times, a list of travel times as directed edges times[i] = (ui, vi, wi), where ui is the source node, vi is the target node, and wi is the time it takes for a signal to travel from source to target.

We will send a signal from a given node k. Return *the* ***minimum*** *time it takes for all the* n *nodes to receive the signal*. If it is impossible for all the n nodes to receive the signal, return -1.

**Example 1:**



Input: times = [[2,1,1],[2,3,1],[3,4,1]], n = 4, k = 2

Output: 2

**Example 2:**

Input: times = [[1,2,1]], n = 2, k = 1

Output: 1

**Example 3:**

Input: times = [[1,2,1]], n = 2, k = 2

Output: -1

**Constraints:**

1 <= k <= n <= 100

1 <= times.length <= 6000

times[i].length == 3

1 <= ui, vi <= n

ui != vi

0 <= wi <= 100

All the pairs (ui, vi) are **unique**. (i.e., no multiple edges.)

**Attempt 1: 2022-11-20**

**Solution 1:  Find minimum distance in a Directed & Weighted Graph using BFS [Dijkstra's algorithm] (120min)**

**Style 1: With "visited" array, we don't really need to maintain "visited" array in below Dijkstra algorithm is an immature solution, refer to** [Dijkstra Shortest Path Algorithm - A Detailed and Visual Introduction](note://80857119213E49EC840091BB3F7E4356) **for detail reason**

class Solution {

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

        // Build graph

        Map<Integer, List<int[]>> graph = new HashMap<Integer, List<int[]>>();

        for(int i = 1; i <= n; i++) {

            graph.put(i, new ArrayList<int[]>());

        }

        for(int[] time : times) {

            graph.get(time[0]).add(new int[]{time[1], time[2]});

        }

        // Dijkstra with minimum priority queue

        // minPQ -> int[]{from, distance}

        PriorityQueue<int[]> minPQ = new PriorityQueue<int[]>((a, b) -> a[1] - b[1]);

        boolean[] visited = new boolean[n + 1];

        // Record minimum distance between node k to each node, to find minimum

        // distance, initially with maximum value

        int[] distances = new int[n + 1];

        Arrays.fill(distances, Integer.MAX\_VALUE);

        // Since label start from 1, no need 0

        distances[0] = 0;

        // The initial start point is node k, distance for node k to itself is 0

        distances[k] = 0;

        minPQ.offer(new int[]{k, 0});

        while(!minPQ.isEmpty()) {

            int[] cur = minPQ.poll();

            int from = cur[0];

            int dist = cur[1];

            if(visited[from]) {

                continue;

            }

            n--;

            visited[from] = true;

            for(int[] neighbour : graph.get(from)) {

                int targetnode = neighbour[0];

                int curnodeToTargetnodeDistance = neighbour[1];

                int newDist = Math.min(distances[targetnode], curnodeToTargetnodeDistance + dist);

                // Update distance record for neighbour node

                distances[targetnode] = newDist;

                minPQ.offer(new int[]{targetnode, newDist});

            }

        }

        // Condition to complete Dijkstra algorithm: Able to visit all nodes

        // n == 0 means able to visit all nodes from node k

        if(n == 0) {

            // Find the maximum distance among all path start from node k to other nodes,

            // this maximum distance is the minimum time it takes for all n nodes to

            // receive the signal

            int maxDistance = 0;

            for(int d : distances) {

                maxDistance = Math.max(maxDistance, d);

            }

            return maxDistance;

        } else {

            return -1;

        }

    }

}

Complexity Analysis

Here N is the number of nodes and E is the number of total edges in the given network.

Time complexity: O(N+ElogN)

Dijkstra's Algorithm takes O(ElogN). Finding the minimum time required in times takes O(N).

The maximum number of vertices that could be added to the priority queue is E. Thus, push

and pop operations on the priority queue take O(logE) time. The value of E can be at most N⋅(N−1).

Therefore, O(logE) is equivalent to O(logN^2) which in turn equivalent to O(2⋅logN).

Hence, the time complexity for priority queue operations equals O(logN).

Although the number of vertices in the priority queue could be equal to E, we will only visit

each vertex only once. If we encounter a vertex for the second time, then curnodeToTargetnodeDistance

will be greater than times[currNode], and we can continue to the next vertex in the priority queue.

Hence, in total E edges will be traversed and for each edge, there could be one priority queue

insertion operation. Hence, the time complexity is equal to O(N+ElogN).

Space complexity: O(N+E)

Building the adjacency list will take O(E) space. Dijkstra's algorithm takes O(E) space for

priority queue because each vertex could be added to the priority queue N - 1N−1 time which

makes it N∗(N−1) and O(N^2) is equivalent to O(E). times takes O(N) space.

**Style 2: Without "visited" array, standard Dijkstra Algorithm**

class Solution {

    public int networkDelayTime(int[][] times, int N, int K) {

        // Build graph

        List<List<int[]>> graph = new ArrayList<>();

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

            graph.add(new ArrayList<>());

        }

        for (int[] edge : times) {

            int fr\_node = edge[0];

            int to\_node = edge[1];

            int cost = edge[2];

            graph.get(fr\_node).add(new int[]{to\_node, cost});

        }

        // Record minimum distance between node k to each node, to find minimum

        // distance, initially with maximum value

        int[] distances = new int[N + 1];

        Arrays.fill(distances, Integer.MAX\_VALUE);

        // Dijkstra with minimum priority queue

        // minPQ -> int[]{from, distance}

        PriorityQueue<int[]> minPQ = new PriorityQueue<>((a, b) -> a[1] - b[1]);

        // The initial start point is node k, distance for node k to itself is 0

        distances[K] = 0;

        minPQ.offer(new int[]{K, 0});

        while (!minPQ.isEmpty()) {

            int[] cur = minPQ.poll();

            int curNode = cur[0];

            int curCost = cur[1];

            for (int[] neighbor : graph.get(curNode)) {

                int newCost = distances[curNode] + neighbor[1];

                if (newCost < distances[neighbor[0]]) {

                    distances[neighbor[0]] = newCost;

                    minPQ.offer(new int[]{neighbor[0], newCost});

                }

            }

        }

        int max\_time = Integer.MIN\_VALUE;

        for (int i = 1; i < distances.length; ++i) {

            if (max\_time < distances[i]) {

                max\_time = distances[i];

            }

        }

        return max\_time == Integer.MAX\_VALUE ? -1 : max\_time;

    }

}

**Refer to**

<https://leetcode.com/problems/network-delay-time/solutions/340477/c-dijkstra-with-priority-queue/>

class Solution {

public:

int networkDelayTime(vector<vector<int>>& times, int N, int K) {

// build graph

vector<vector<pair<int, int> > > graph(N+1);

for (auto edge : times) {

int fr\_node = edge[0];

int to\_node = edge[1];

int cost = edge[2];

graph[fr\_node].push\_back(make\_pair(cost, to\_node));

}

vector<int> dist(N+1, INT\_MAX);

dist[K] = 0;

pq.push(make\_pair(0, K));

while (!pq.empty()) {

pair<int,int> x = pq.top();

pq.pop();

for (auto neighbor : graph[x.second]) {

int ar = dist[x.second] + neighbor.first;

if (ar < dist[neighbor.second]) {

dist[neighbor.second] = ar;

pq.push(make\_pair(ar, neighbor.second));

}

}

}

int max\_time = INT\_MIN;

for (int i = 1; i < dist.size(); ++i) {

if (max\_time < dist[i]) {

max\_time = dist[i];

}

}

return max\_time == INT\_MAX? -1 : max\_time;

}

};

**Style 3: Without "visited" array, also early terminate when same node process again with Dijkstra Algorithm**

class Solution {

    public int networkDelayTime(int[][] times, int N, int K) {

        // Build graph

        List<List<int[]>> graph = new ArrayList<>();

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

            graph.add(new ArrayList<>());

        }

        for (int[] edge : times) {

            int fr\_node = edge[0];

            int to\_node = edge[1];

            int cost = edge[2];

            graph.get(fr\_node).add(new int[]{to\_node, cost});

        }

        // Record minimum distance between node k to each node, to find minimum

        // distance, initially with maximum value

        int[] distances = new int[N + 1];

        Arrays.fill(distances, Integer.MAX\_VALUE);

        // Dijkstra with minimum priority queue

        // minPQ -> int[]{from, distance}

        PriorityQueue<int[]> minPQ = new PriorityQueue<>((a, b) -> a[1] - b[1]);

        // The initial start point is node k, distance for node k to itself is 0

        distances[K] = 0;

        minPQ.offer(new int[]{K, 0});

        while (!minPQ.isEmpty()) {

            int[] cur = minPQ.poll();

            int curNode = cur[0];

            int curCost = cur[1];

            // Add below statement improve Elapsed Time from 11ms to 9ms

            // Skip if encounter same node again and node's distance is outdated:

            // Once a node is processed earlier, no shorter distance can be found

            // for it due to the PriorityQueue. The distances[] array acts as a

            // safeguard: it prevents processing outdated or longer paths,

            // eliminating the need for a visited array. If a node is encountered

            // again in the PriorityQueue, its distance will not be processed

            // since this node's guaranteed smallest distance result (based on

            // Minimum PriorityQueue natruality) stored at distances[curNode]

            // earlier, ensured by the condition below:

            // "if (curCost > distances[curNode]) continue;"

            // Additionally, if not add this statement, the Dijkstra algorithm

            // still works, only won't terminate early when encountering same node

            // again, and same node will be added onto PriorityQueue again and

            // all further process blocked later with condition below:

            // "if (newCost < distances[neighbor[0]]) {...}"

            // since the newCost is surely >= distances[neighbor[0]]

            // at a certain moment

            if (curCost > distances[curNode]) {

                continue;

            }

            for (int[] neighbor : graph.get(curNode)) {

                int newCost = distances[curNode] + neighbor[1];

                if (newCost < distances[neighbor[0]]) {

                    distances[neighbor[0]] = newCost;

                    minPQ.offer(new int[]{neighbor[0], newCost});

                }

            }

        }

        int max\_time = Integer.MIN\_VALUE;

        for (int i = 1; i < distances.length; ++i) {

            if (max\_time < distances[i]) {

                max\_time = distances[i];

            }

        }

        return max\_time == Integer.MAX\_VALUE ? -1 : max\_time;

    }

}

**Dijkstra algorithm no need "visited" array ? why ?**

**Refer to chatGPT**

In Dijkstra's algorithm, a **visited** array is often not explicitly required because the algorithm inherently ensures that each node is processed only once. This is achieved by the priority queue (or min-heap) and the way distances are updated and processed. Let's explore why this is the case:

**1. Priority Queue Guarantees Optimal Order**

Dijkstra's algorithm uses a priority queue to always process the node with the smallest tentative distance first.

**Once a node is dequeued (processed), it means the shortest path to that node is already determined, and its distance cannot be improved further.**

**Reason**: **Relaxation of edges from other nodes cannot produce a smaller distance due to the priority queue's property.**

**Thus, there's no need to revisit a node, which is the primary purpose of a visited array.**

**2. Implicit Handling of Re-visits**

In the case where an already processed node is encountered again (due to a shorter path being found earlier in the queue), the algorithm skips it naturally because:

The node's tentative distance will not be smaller than the distance already processed.

Priority queues ensure that nodes are processed in increasing order of their distances.

This property makes a **visited** array redundant.

**3. Efficiency and Correctness**

Using a **visited** array in Dijkstra’s algorithm can slightly improve performance by preventing unnecessary operations like inserting the same node into the priority queue multiple times.

However, many implementations avoid explicitly maintaining a **visited** array because the priority queue mechanism already handles this aspect implicitly.

**4. When a Visited Array Might Be Useful**

Modified Dijkstra: If the algorithm is modified for specific scenarios, such as finding paths with certain constraints, a **visited** array might help.

Memory Optimization: In dense graphs, using a priority queue without a visited array could lead to more nodes being inserted into the queue than necessary, causing memory overhead.

**Example Without Visited Array**

class Dijkstra {

public int[] shortestPath(int n, int[][] edges, int source) {

Map<Integer, List<int[]>> graph = new HashMap<>();

for (int[] edge : edges) {

graph.computeIfAbsent(edge[0], k -> new ArrayList<>()).add(new int[]{edge[1], edge[2]});

graph.computeIfAbsent(edge[1], k -> new ArrayList<>()).add(new int[]{edge[0], edge[2]}); // Undirected graph

}

int[] dist = new int[n];

Arrays.fill(dist, Integer.MAX\_VALUE);

dist[source] = 0;

PriorityQueue<int[]> pq = new PriorityQueue<>(Comparator.comparingInt(a -> a[1])); // {node, distance}

pq.add(new int[]{source, 0});

while (!pq.isEmpty()) {

int[] current = pq.poll();

int node = current[0];

int d = current[1];

// Skip if this distance is outdated

if (d > dist[node]) continue;

for (int[] neighbor : graph.getOrDefault(node, new ArrayList<>())) {

int next = neighbor[0];

int weight = neighbor[1];

if (dist[node] + weight < dist[next]) {

dist[next] = dist[node] + weight;

pq.add(new int[]{next, dist[next]});

}

}

}

return dist; // Returns shortest distances from the source to all nodes

}

}

**Key Insight**

The algorithm skips nodes that are already processed because the priority queue ensures optimal order. Using a **visited** array becomes optional but redundant for correctness.

**Can you use Leetcode 743 to elaborate how Dijkstra works without using visited array ?**

**Key Idea**

Use a **priority queue (min-heap)** to process nodes in order of their smallest distance.

Maintain a dist[] array to store the shortest known distance to each node.

Initialize all distances to infinity (Integer.MAX\_VALUE), except for the starting node (k), which is 0.

Each node is processed only once at its shortest distance because:

If a shorter path to a node is found, it will re-enter the priority queue.

Priority queue ensures that only the shortest distance is processed first, making re-visiting unnecessary.

**Implementation Without Visited Array**

class Solution {

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

// Step 1: Build the graph

Map<Integer, List<int[]>> graph = new HashMap<>();

for (int[] edge : times) {

graph.computeIfAbsent(edge[0], x -> new ArrayList<>()).add(new int[]{edge[1], edge[2]});

}

// Step 2: Initialize distance array

int[] dist = new int[n + 1]; // 1-indexed nodes

Arrays.fill(dist, Integer.MAX\_VALUE);

dist[k] = 0; // Distance to source is 0

// Step 3: Priority queue to process nodes by shortest distance

PriorityQueue<int[]> pq = new PriorityQueue<>(Comparator.comparingInt(a -> a[1])); // {node, distance}

pq.add(new int[]{k, 0}); // Start with the source node

// Step 4: Process the queue

while (!pq.isEmpty()) {

int[] current = pq.poll();

int node = current[0];

int time = current[1];

// Skip if the distance is outdated

if (time > dist[node]) continue;

// Traverse all neighbors

for (int[] neighbor : graph.getOrDefault(node, new ArrayList<>())) {

int nextNode = neighbor[0];

int weight = neighbor[1];

// If a shorter path to the neighbor is found

if (dist[node] + weight < dist[nextNode]) {

dist[nextNode] = dist[node] + weight;

pq.add(new int[]{nextNode, dist[nextNode]});

}

}

}

// Step 5: Find the maximum distance to determine the result

int maxTime = 0;

for (int i = 1; i <= n; i++) {

if (dist[i] == Integer.MAX\_VALUE) return -1; // Unreachable node

maxTime = Math.max(maxTime, dist[i]);

}

return maxTime;

}

}

**How It Works Without Visited Array**

Priority Queue Ensures Shortest Path Processing

Nodes are processed in increasing order of distance.

When a node is dequeued, it is guaranteed to be at its shortest distance because:

All shorter paths have been processed already.

**If a node is encountered again in the priority queue, its distance will not be processed if it’s already smaller (ensured by the condition if (time > dist[node]) continue).**

No Need to Explicitly Track Visited Nodes

Once a node is processed, no shorter distance can be found for it due to the priority queue.

The dist[] array acts as a safeguard: it prevents processing outdated or longer paths, eliminating the need for a visited array.

**Key Takeaways**

The **priority queue** and dist[] array ensure that nodes are processed only at their shortest distances.

**No explicit visited array is needed because outdated paths are skipped naturally by checking if (time > dist[node]).**

The algorithm has a time complexity of O((V+E) log⁡V), where V is the number of nodes and E is the number of edges.

**Solution 2:  Promote by removing distances array (10min)**

class Solution {

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

        int result = 0;

        // Build graph

        Map<Integer, List<int[]>> graph = new HashMap<Integer, List<int[]>>();

        for(int i = 1; i <= n; i++) {

            graph.put(i, new ArrayList<int[]>());

        }

        for(int[] time : times) {

            graph.get(time[0]).add(new int[]{time[1], time[2]});

        }

        // Dijkstra with minimum priority queue

        // minPQ -> int[]{from, distance}

        PriorityQueue<int[]> minPQ = new PriorityQueue<int[]>((a, b) -> a[1] - b[1]);

        boolean[] visited = new boolean[n + 1];

        minPQ.offer(new int[]{k, 0});

        while(!minPQ.isEmpty()) {

            int[] cur = minPQ.poll();

            int from = cur[0];

            int dist = cur[1];

            if(visited[from]) {

                continue;

            }

            n--;

            visited[from] = true;

            result = dist;

            for(int[] neighbour : graph.get(from)) {

                int targetnode = neighbour[0];

                int curnodeToTargetnodeDistance = neighbour[1];

                minPQ.offer(new int[]{targetnode, curnodeToTargetnodeDistance + dist});

            }

        }

        return n == 0 ? result : -1;

    }

}

Complexity Analysis

Here N is the number of nodes and E is the number of total edges in the given network.

Time complexity: O(N+ElogN)

Dijkstra's Algorithm takes O(ElogN). Finding the minimum time required in times takes O(N).

The maximum number of vertices that could be added to the priority queue is E. Thus, push

and pop operations on the priority queue take O(logE) time. The value of E can be at most N⋅(N−1).

Therefore, O(logE) is equivalent to O(logN^2) which in turn equivalent to O(2⋅logN).

Hence, the time complexity for priority queue operations equals O(logN).

Although the number of vertices in the priority queue could be equal to E, we will only

visit each vertex only once. If we encounter a vertex for the second time, then curnodeToTargetnodeDistance

will be greater than times[currNode], and we can continue to the next vertex in the priority queue.

Hence, in total E edges will be traversed and for each edge, there could be one priority queue

insertion operation. Hence, the time complexity is equal to O(N+ElogN).

Space complexity: O(N+E)

Building the adjacency list will take O(E) space. Dijkstra's algorithm takes O(E) space for

priority queue because each vertex could be added to the priority queue N - 1N−1 time which

makes it N∗(N−1) and O(N^2) is equivalent to O(E). times takes O(N) space.

**Refer to**

<https://leetcode.com/problems/network-delay-time/discuss/210698/Java-Djikstrabfs-Concise-and-very-easy-to-understand>

I think bfs and djikstra are very similar problems. It's just that djikstra cost is different compared with bfs, so use priorityQueue instead a Queue for a standard bfs search.

class Solution {

    public int networkDelayTime(int[][] times, int N, int K) {

        Map<Integer, Map<Integer,Integer>> map = new HashMap<>();

        for(int[] time : times){

            map.putIfAbsent(time[0], new HashMap<>());

            map.get(time[0]).put(time[1], time[2]);

        }

        //distance, node into pq

        Queue<int[]> pq = new PriorityQueue<>((a,b) -> (a[0] - b[0]));

        pq.add(new int[]{0, K});

        boolean[] visited = new boolean[N+1];

        int res = 0;

        while(!pq.isEmpty()){

            int[] cur = pq.remove();

            int curNode = cur[1];

            int curDist = cur[0];

            if(visited[curNode]) continue;

            visited[curNode] = true;

            res = curDist;

            N--;

            if(map.containsKey(curNode)){

                for(int next : map.get(curNode).keySet()){

                    pq.add(new int[]{curDist + map.get(curNode).get(next), next});

                }

            }

        }

        return N == 0 ? res : -1;

    }

}

**Another promotion:**

Nice code, note one **improvement** which can reduce time from 62ms to 49ms for me: return res; when N = 0, i.e. the code becomes to:

<https://leetcode.com/problems/network-delay-time/discuss/210698/Java-Djikstrabfs-Concise-and-very-easy-to-understand/275555>

You don't have to poll all the elements from pq, you can just terminate it when N = 0, since when N = 0you have visited all the nodes along the shortest path from the source node, all nodes left in the pq are the redundant nodes along the non-shortest path. you can save time complexity of pop operation for O(klogk)

class Solution {

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

        int result = 0;

        // Build graph

        Map<Integer, List<int[]>> graph = new HashMap<Integer, List<int[]>>();

        for(int i = 1; i <= n; i++) {

            graph.put(i, new ArrayList<int[]>());

        }

        for(int[] time : times) {

            graph.get(time[0]).add(new int[]{time[1], time[2]});

        }

        // Dijkstra with minimum priority queue

        // minPQ -> int[]{from, distance}

        PriorityQueue<int[]> minPQ = new PriorityQueue<int[]>((a, b) -> a[1] - b[1]);

        boolean[] visited = new boolean[n + 1];

        minPQ.offer(new int[]{k, 0});

        while(!minPQ.isEmpty()) {

            int[] cur = minPQ.poll();

            int from = cur[0];

            int dist = cur[1];

            if(visited[from]) {

                continue;

            }

            n--;

            visited[from] = true;

            result = dist;

            if(n == 0) {

                return result;

            }

            for(int[] neighbour : graph.get(from)) {

                int targetnode = neighbour[0];

                int curnodeToTargetnodeDistance = neighbour[1];

                minPQ.offer(new int[]{targetnode, curnodeToTargetnodeDistance + dist});

            }

        }

        return -1;

    }

}

**Refer to**

[L505.Lint788.The Maze II (Ref.L490,L743)](note://6441348532EF41BEB2DDC34801D7AA20)

[L1368.Minimum Cost to Make at Least One Valid Path in a Grid (Ref.L2290)](note://WEB4b3aa9003dbf5915537540decf09b434)

[Dijkstra Shortest Path Algorithm - A Detailed and Visual Introduction](note://80857119213E49EC840091BB3F7E4356)