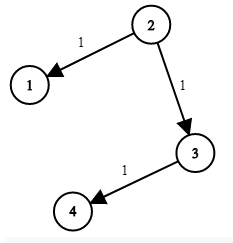
<https://leetcode.com/problems/network-delay-time/>

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;

}

};

**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)

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