Network Delay Time

There are `N` network nodes, labelled `1` to `N`.

Given `times`, a list of travel times as **directed** edges `times[i] = (u, v, w)`, where `u` is the source node, `v` is the target node, and `w` is the time it takes for a signal to travel from source to target.

Now, we send a signal from a certain node `K`. How long will it take for all nodes to receive the signal? If it is impossible, return `-1`.

Note:

1. `N` will be in the range `[1, 100]`.
2. `K` will be in the range `[1, N]`.
3. The length of `times` will be in the range `[1, 6000]`.
4. All edges `times[i] = (u, v, w)` will have `1 and  1 .`

Solution 1

It is a direct graph.

1. Use Map<Integer, Map<Integer, Integer>> to store the source node, target node and the distance between them.
2. Offer the node K to a PriorityQueue.
3. Then keep getting the closest nodes to the current node and calculate the distance from the source (K) to this node (absolute distance). Use a Map to store the shortest absolute distance of each node.
4. Return the node with the largest absolute distance.

```
public int networkDelayTime(int[][] times, int N, int K) {
        if(times == null || times.length == 0){
            return -1;
        }
        // store the source node as key. The value is another map of the neighbor no
des and distance.
        Map<Integer, Map<Integer, Integer>> path = new HashMap<>();
        for(int[] time : times){
            Map<Integer, Integer> sourceMap = path.get(time[0]);
            if(sourceMap == null){
                sourceMap = new HashMap<>();
                path.put(time[0], sourceMap);
            }
            Integer dis = sourceMap.get(time[1]);
            if(dis == null || dis > time[2]){
                sourceMap.put(time[1], time[2]);
            }

        }

        //Use PriorityQueue to get the node with shortest absolute distance
        //and calculate the absolute distance of its neighbor nodes.
        Map<Integer, Integer> distanceMap = new HashMap<>();
        distanceMap.put(K, 0);
        PriorityQueue<int[]> pq = new PriorityQueue<>((i1, i2) -> {return i1[1] - i2
[1];});
        pq.offer(new int[]{K, 0});
        int max = -1;
        while(!pq.isEmpty()){
            int[] cur = pq.poll();
            int node = cur[0];
            int distance = cur[1];

            // Ignore processed nodes
            if(distanceMap.containsKey(node) && distanceMap.get(node) < distance){
                continue;
            }

            Map<Integer, Integer> sourceMap = path.get(node);
            if(sourceMap == null){
                continue;
            }
            for(Map.Entry<Integer, Integer> entry : sourceMap.entrySet()){
                int absoluteDistence = distance + entry.getValue();
                int targetNode = entry.getKey();
                if(distanceMap.containsKey(targetNode) && distanceMap.get(targetNod
e) <= absoluteDistence){
                    continue;
                }
                distanceMap.put(targetNode, absoluteDistence);
                pq.offer(new int[]{targetNode, absoluteDistence});
            }
        }
        // get the largest absolute distance.
        for(int val : distanceMap.values()){
            if(val > max){
                max = val;
            }
        }
        return distanceMap.size() == N ? max : -1;
}
```

written by Samuel.Y.T.Ji original link here

Solution 2

1. `shortest-path`:The delay of the network is the time for the signal to reach the furthest node, so this is a `shortest-path` problem, and we need to calculate `shortest-path` for all nodes;
2. `initialization`: We can initialize the `wait time` for all nodes to `infinity`, except the source `K`, its `wait time` is `0`. Then start from `K`, we update the `best wait time` for all its neighbor `nb` as `delay[K] + w(K, nb)`, note that these are temporary best result for these neighbors, their `best-wait-time` can be further optimized by `RELAXATION` from other incoming edges later.
3. `RELAXATION`: by definition is - for node `v`, if there is an edge `(u, v)` from `u` to `v`, its distance-to-source `dist(v)` is longer than `dist(u) + w(u, v)`, where `w(u,v)` is the length(or weight) of edge `(u,v)`, then `dist(v)` can be optimized as `dist(u) + w(u, v)`.
4. `level order BFS`: what we are doing here is, first `RELAX` all the neighbors of `K`, then `RELAX` all the neighbors of its neighbors that `are updated with a better wait-time`, then keep going ... until all node can no longer be `RELAX`ed.
5. `next level` - note that the candidates for next level can be duplicated, because nodes in the current level could share neighbors. That's why we want to use an `set` to keep track of all the neighbors ever `RELAX`-ed in this level. So, don't put neighbor in the queue for next round if it is already visited.

`NOTE`:
`negative weight cycle` - Cycles can be avoided as long as its weight will be not less than without the cycle. But this method cannot handle `negative weight cycle`.
BFS
Java - adjacency matrix

```java
class Solution {
    public int networkDelayTime(int[][] times, int N, int K) {
        int[] delay = new int[N + 1];
        Arrays.fill(delay, Integer.MAX_VALUE);
        Integer[][] edge = new Integer[101][101];
        for (int[] e : times) edge[e[0]][e[1]] = e[2];
        Queue<Integer> q = new LinkedList<>();
        q.offer(K);
        delay[K] = 0;
        while (!q.isEmpty()) {
            Set<Integer> set = new HashSet<>();
            for (int n = q.size(); n > 0; n--) {
                int u = q.poll();
                for (int v = 1; v <= 100; v++) {
                    if (edge[u][v] != null && delay[u] + edge[u][v] < delay[v]) {
                        if (!set.contains(v)) {
                            set.add(v);
                            q.offer(v);
                        }
                        delay[v] = delay[u] + edge[u][v];
                    }
                }
            }
        }
        int maxdelay = 0;
        for (int i = 1; i <= N; i++)
            maxdelay = Math.max(maxdelay, delay[i]);

        return maxdelay == Integer.MAX_VALUE ? -1 : maxdelay;
    }
}
```

C++

```cpp
class Solution {
public:
    int networkDelayTime(vector<vector<int>>& times, int N, int K) {
        vector<int> waits(N + 1, INT_MAX);
        map<int, map<int, int>> adj;
        for (auto e : times) adj[e[0]][e[1]] = e[2];
        queue<int> q;
        q.push(K);
        waits[K] = 0;
        while (!q.empty()) {
            set<int> set;
            for (int n = q.size(); n > 0; n--) {
                int u = q.front(); q.pop();
                for (pair<int, int> nb : adj[u]) {
                    int v = nb.first;
                    if (waits[u] + adj[u][v] < waits[v]) {
                        if (!set.count(v)) {
                            set.insert(v);
                            q.push(v);
                        }
                        waits[v] = waits[u] + adj[u][v];
                    }
                }
            }
        }
        int maxwait = 0;
        for (int i = 1; i <= N; i++)
            maxwait = max(maxwait, waits[i]);
        return maxwait == INT_MAX ? -1 : maxwait;
    }
};
```

Bellman Ford

C++

```cpp
class Solution {
public:
    int networkDelayTime(vector<vector<int>>& times, int N, int K) {
        vector<int> dist(N + 1, INT_MAX);
        dist[K] = 0;
        for (int i = 0; i < N; i++) {
            for (vector<int> e : times) {
                int u = e[0], v = e[1], w = e[2];
                if (dist[u] != INT_MAX && dist[v] > dist[u] + w) {
                    dist[v] = dist[u] + w;
                }
            }
        }

        int maxwait = 0;
        for (int i = 1; i <= N; i++)
            maxwait = max(maxwait, dist[i]);
        return maxwait == INT_MAX ? -1 : maxwait;
    }
};
```

written by alexander original link here

Solution 3

The idea is to find the time to reach every other node from given node K
Then, to check if all nodes can be reached and if it can be reached then return the time taken to reach the farthest node (node which take longest to get the signal).
As the signal traverses concurrently to all nodes, we have to find the maximum time it takes to reach a node among all nodes from given node K.
If any single node takes Integer.MAX_Value, then return -1, as not all nodes can be reached

add the index of all the nodes which can be reached from a node to a list and store this list in a hashmap

then its similar to dijkstra's shortest path algorithm except that I am not using Priority Queue to get the minimum distance node. See comments.

```java
class Solution {
    public int networkDelayTime(int[][] times, int N, int K) {
        int r = times.length, max = Integer.MAX_VALUE;

        Map<Integer,List<Integer>> map = new HashMap<>();
        for(int i=0;i<r;i++){
            int[] nums = times[i];
            int u = nums[0];
            int v = nums[1];
            List<Integer> list = map.getOrDefault(u,new ArrayList<>());

            list.add(i);

            map.put(u,list);
        }
        if(map.get(K) == null){
            return -1;// no immediate neighbor of node K, so return -1
        }
        int[] dist = new int[N+1];//dist[i] is the time taken to reach node i from n
ode k
        Arrays.fill(dist,max);

        dist[K] = 0;
        Queue<Integer> queue = new LinkedList<>();

        queue.add(K);

        while(!queue.isEmpty()){
            int u = queue.poll();
            int t = dist[u];
            List<Integer> list = map.get(u);// get the indices of all the neighbors
of node u
            if(list == null)
                continue;

            for(int n:list){
                int v = times[n][1];
                int time = times[n][2];// time taken to reach from u to v
                 if(dist[v] > t + time){// if time taken to reach v from k is great
er than time taken to reach from k to u + time taken to reach from u to v , then upda
er than time taken to reach from K to u + time taken to reach from u to v, then upda
te dist[v]
                    dist[v] = t + time;
                    queue.add(v);// as we have found shorter distance to node v, ex
plore all neighbors of v
                }
            }
        }

        int res = -1;

        for(int i=1;i<=N;i++){
            int d = dist[i];
            if(d == max){// if d is max, it means node i can not be reached from K,
so return -1
                return -1;
            }
            res = d > res ? d : res;
        }

        return res;
    }
}
```

Note: I have updated the solution description and solution. Yes, my solution is not exactly dijkstra, but it was inspired by it and I thought it was very similar except that it is not very similar.

written by ashish53v original link here