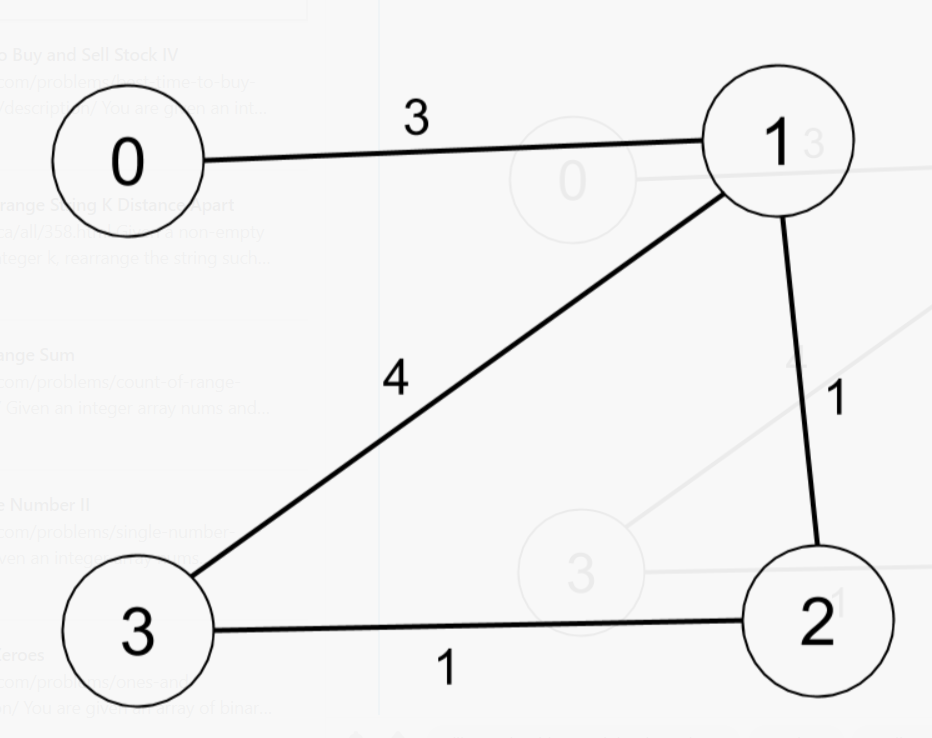
<https://leetcode.com/problems/find-the-city-with-the-smallest-number-of-neighbors-at-a-threshold-distance/description/>

There are n cities numbered from 0 to n-1. Given the array edges where edges[i] = [fromi, toi, weighti] represents a bidirectional and weighted edge between cities fromi and toi, and given the integer distanceThreshold.

Return the city with the smallest number of cities that are reachable through some path and whose distance is **at most** distanceThreshold, If there are multiple such cities, return the city with the greatest number.

Notice that the distance of a path connecting cities ***i*** and ***j*** is equal to the sum of the edges' weights along that path.

**Example 1:**



Input: n = 4, edges = [[0,1,3],[1,2,1],[1,3,4],[2,3,1]], distanceThreshold = 4

Output: 3

Explanation: The figure above describes the graph.

The neighboring cities at a distanceThreshold = 4 for each city are:

City 0 -> [City 1, City 2]

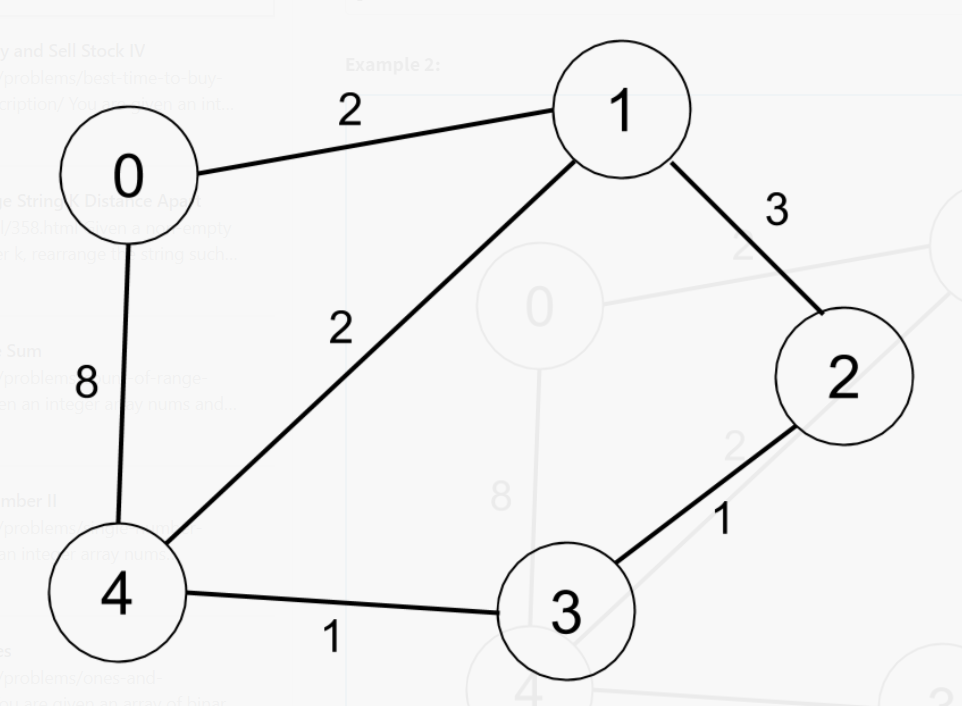
City 1 -> [City 0, City 2, City 3]

City 2 -> [City 0, City 1, City 3]

City 3 -> [City 1, City 2]

Cities 0 and 3 have 2 neighboring cities at a distanceThreshold = 4, but we have to return city 3 since it has the greatest number.

**Example 2:**



Input: n = 5, edges = [[0,1,2],[0,4,8],[1,2,3],[1,4,2],[2,3,1],[3,4,1]], distanceThreshold = 2

Output: 0

Explanation: The figure above describes the graph.

The neighboring cities at a distanceThreshold = 2 for each city are:

City 0 -> [City 1]

City 1 -> [City 0, City 4]

City 2 -> [City 3, City 4]

City 3 -> [City 2, City 4]

City 4 -> [City 1, City 2, City 3]

The city 0 has 1 neighboring city at a distanceThreshold = 2.

**Constraints:**

* 2 <= n <= 100
* 1 <= edges.length <= n \* (n - 1) / 2
* edges[i].length == 3
* 0 <= fromi < toi < n
* 1 <= weighti, distanceThreshold <= 10^4
* All pairs (fromi, toi) are distinct.

**Attempt 1: 2023-10-29**

**Solution 1: Dijkstra (120 min)**

**Style 1: MaxPQ**

class Solution {

public int findTheCity(int n, int[][] edges, int distanceThreshold) {

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

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

graph.put(i, new HashSet<>());

}

for(int[] edge : edges) {

if(edge[2] > distanceThreshold) {

continue;

}

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

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

}

int min = n + 1;

int result = -1;

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

int count = bfs(i, graph, distanceThreshold);

if(count <= min) {

min = count;

result = i;

}

}

return result;

}

private int bfs(int i, Map<Integer, Set<int[]>> graph, int distanceThreshold) {

int count = -1;

Set<Integer> visited = new HashSet<>();

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

maxPQ.offer(new int[]{i, distanceThreshold});

while(!maxPQ.isEmpty()) {

int[] cur = maxPQ.poll();

if(visited.contains(cur[0])) {

continue;

}

visited.add(cur[0]);

count++;

for(int[] neighbour : graph.get(cur[0])) {

if(!visited.contains(neighbour[0]) && cur[1] >= neighbour[1]) {

maxPQ.offer(new int[]{neighbour[0], cur[1] - neighbour[1]});

}

}

}

return count;

}

}

Dijkstra:

For each node A

Calculate the minimum distance from node A to every other node

Find how many nodes are reachable within the distanceThreshold

Output the last node with least reachability

Complexity

|E| = (N \* (N - 1)) / 2

|V| = N

Time:

Dijkstra has O(|E| \* log(|V|)) and Dijkstra is applied to each node (i.e. |V| times).

O(|V| \* |E| \* log(|V|)) = O(N^3 \* log(N))

Space:

O(|V| + |E|) for the graph adjacency list

O(|V|) for visited and distance arrays used by dijkstra, and O(N) for priority queue

O(|V| + |E|) = O(N^2)

**The Reason of DFS Not Working (Explain Graph and Example)**

**Refer to**

<https://leetcode.com/problems/find-the-city-with-the-smallest-number-of-neighbors-at-a-threshold-distance/solutions/490555/the-reason-of-dfs-not-working-explain-graph-and-example/>

Many people solve this problem with a DFS algorithm and have a problem with some very big test cases. I found a short test case will fail the DFS solutions and explain the reason.

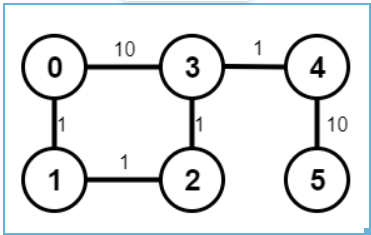
This test case will fail a DFS solution(I already contributed this test case):

6

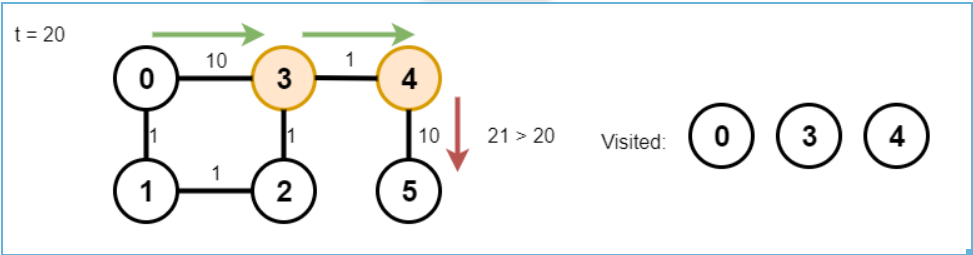
{{0,1,1},{1,2,1},{2,3,1},{0,3,10},{3,4,1},{4,5,10}}

20

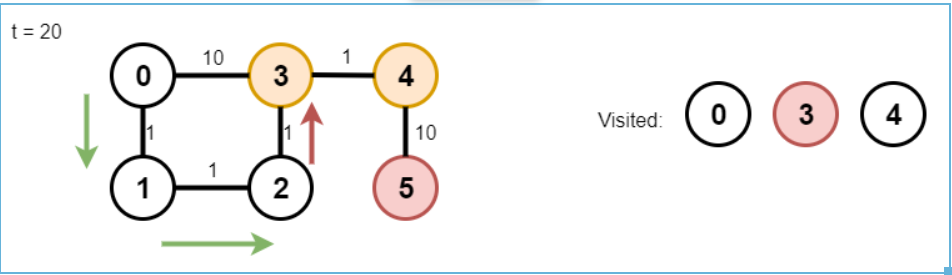
The graph looks like this, and The **distanceThreshold = 20**



If your DFS go though the **0->3** before the **0->1** the error will encounter. Note the visited list.



The next time your DFS goes **0->1** it will find the 3 already visited and the 5 will never visit. So that's why those case will have fewer city visit than expect.



By the way, this question is basiclly a trap. If you never heard of **Floyd-Warshall algorithm**or noticed those cases. A lot people will directly use DFS. This question's contest AC rate only: **0.259**

**在使用了max PriorityQueue以后下面我们测试以上给出的图例：**

6

{{0,1,1},{1,2,1},{2,3,1},{0,3,10},{3,4,1},{4,5,10}}

20

**我们可以发现基于BFS的Dijkstra可以成功回避从0 -> 3的路径错误路径选择，因为在每次首先选择当前节点到目标节点的最短路径最优先的情况下，当前节点0到目标节点3的最短路径是0 -> 1 -> 2 -> 3而非0 -> 3，而这一点可以通过引入PriorityQueue实现（根据题意这里选择了max PriorityQueue，因为比较的是基于给定的定值distanceThreshold，在走过每一段当前节点到目标节点的路径后还剩多少，剩的越多，当前路径消耗的越少，路径越优先，那么需要知道哪条路径剩的越多，我们可以用max PriorityQueue处理）**

import java.util.\*;

public class Solution {

public int findTheCity(int n, int[][] edges, int distanceThreshold) {

Map<Integer, Set<int[]>> g = new HashMap<>();

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

g.put(i, new HashSet<>());

}

for (int[] e : edges) {

if (e[2] > distanceThreshold) continue;

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

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

}

int min = n + 1;

int res = -1;

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

int count = bfs(g, distanceThreshold, i);

if (count <= min) {

min = count;

res = i;

}

}

return res;

}

private int bfs(Map<Integer,Set<int[]>> g, int distanceThreshold, int i) {

// Queue<int[]> q = new LinkedList<>();

Queue<int[]> q = new PriorityQueue<>((a, b) -> (b[1] - a[1])); // have to be heap over FIFO, and it must be maximum heap

q.add(new int[]{i, distanceThreshold});

Set<Integer> visited = new HashSet<>();

int count = -1;

while (!q.isEmpty()) {

int[] city = q.poll();

if (visited.contains(city[0])) continue;

visited.add(city[0]);

count++;

for (int[] neighbor : g.get(city[0])) {

if (!visited.contains(neighbor[0]) && city[1] >= neighbor[1]) {

q.add(new int[]{neighbor[0], city[1] - neighbor[1]});

}

}

}

return count;

}

public static void main(String[] args) {

Solution so = new Solution();

int[][] edges = new int[][]{{0,1,1},{1,2,1},{2,3,1},{0,3,10},{3,4,1},{4,5,10}};

int result = so.findTheCity(6, edges, 20);

System.out.println(result);

}

}

**Refer to**

<https://leetcode.com/problems/find-the-city-with-the-smallest-number-of-neighbors-at-a-threshold-distance/solutions/490586/why-dijkstra-over-bfs-why-heap-over-fifo/>

Idea:

1. Record the edges and weights by map
2. Use Dijkstra algorithm to traverse every vertex

class Solution {

public int findTheCity(int n, int[][] edges, int distanceThreshold) {

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

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

g.put(i, new HashMap<>());

}

for (int[] e : edges) {

g.get(e[0]).put(e[1], e[2]);

g.get(e[1]).put(e[0], e[2]);

}

int min = n + 1;

int res = -1;

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

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

q.add(new int[]{i, distanceThreshold});

Set<Integer> visited = new HashSet<>();

int count = 0;

while (!q.isEmpty()) {

int[] city = q.poll();

if (!visited.contains(city[0])) {

visited.add(city[0]);

count++;

} else {

continue;

}

Map<Integer, Integer> m = g.get(city[0]);

for (int neighbor : m.keySet()) {

if (!visited.contains(neighbor) && city[1] >= m.get(neighbor)) {

q.add(new int[]{neighbor, city[1] - m.get(neighbor)});

}

}

}

if (count - 1 <= min) {

min = count - 1;

res = i;

}

}

return res;

}

}

**Style 2: MinPQ**

**稍微改动一下MaxPQ就可以变成MinPQ的写法**

class Solution {

public int findTheCity(int n, int[][] edges, int distanceThreshold) {

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

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

graph.put(i, new HashSet<>());

}

for(int[] edge : edges) {

if(edge[2] > distanceThreshold) {

continue;

}

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

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

}

int min = n + 1;

int result = -1;

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

int count = bfs(i, graph, distanceThreshold);

if(count <= min) {

min = count;

result = i;

}

}

return result;

}

private int bfs(int i, Map<Integer, Set<int[]>> graph, int distanceThreshold) {

int count = -1;

Set<Integer> visited = new HashSet<>();

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

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

while(!minPQ.isEmpty()) {

int[] cur = minPQ.poll();

if(visited.contains(cur[0])) {

continue;

}

visited.add(cur[0]);

count++;

for(int[] neighbour : graph.get(cur[0])) {

if(!visited.contains(neighbour[0]) && cur[1] + neighbour[1] <= distanceThreshold) {

minPQ.offer(new int[]{neighbour[0], cur[1] + neighbour[1]});

}

}

}

return count;

}

}

Dijkstra:

For each node A

Calculate the minimum distance from node A to every other node

Find how many nodes are reachable within the distanceThreshold

Output the last node with least reachability

Complexity

|E| = (N \* (N - 1)) / 2

|V| = N

Time:

Dijkstra has O(|E| \* log(|V|)) and Dijkstra is applied to each node (i.e. |V| times).

O(|V| \* |E| \* log(|V|)) = O(N^3 \* log(N))

Space:

O(|V| + |E|) for the graph adjacency list

O(|V|) for visited and distance arrays used by dijkstra, and O(N) for priority queue

O(|V| + |E|) = O(N^2)