1334. Find the City With the Smallest Number of Neighbors at a Threshold Distance

Bibek Timilsina

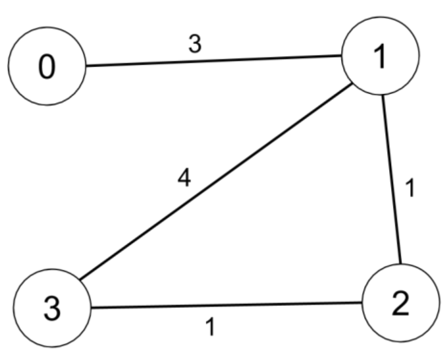
July 13,2024

Problem Statement:

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.

Examples

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]

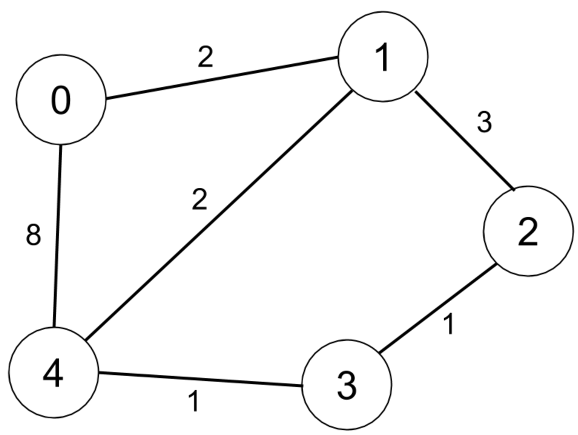
City 1 -> [City 0, City 2, City 3]

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.

Approach for Dijkstra's algorithm:

To solve the problem of finding the city with the smallest number of reachable cities within a given distance threshold using Dijkstra's algorithm, follow these steps:

Step-by-Step Approach

1) **Graph Representation**:
    a) Use an adjacency list to store the graph. Each city will have a list of pairs representing the connected cities and the distances to them.
2) **Dijkstra's Algorithm Implementation**:
    a) Initialize a distance array to store the shortest distance from the starting city to each other city. Set all distances to infinity (`INT_MAX`), except the starting city which is set to `0`.
    b) Use a priority queue (min-heap) to process the cities based on the shortest known distance.
    c) For the current city, update the distances to its neighboring cities if a shorter path is found.
3) **Iterate and Count Reachable Cities**:
    a) For each city, run Dijkstra's algorithm to get the shortest distances to all other cities.
    b) Count how many cities have a distance less than or equal to the distance threshold.
4) **Track Minimum Reachable Cities**:
    a) Keep track of the minimum number of reachable cities and the corresponding city index. If there is a tie, choose the city with the larger index.

Solution Code for Dijkstra's algorithm:

```
#include <iostream>
#include <vector>
#include <queue>
#include <limits.h>
#include <algorithm>

using namespace std;

typedef pair<int, int> pii;

class Solution {
public:
    vector<vector<pii>> graph;

    vector<int> dijkstra(int n, int start, int distanceThreshold) {
        vector<int> distances(n, INT_MAX);
        distances[start] = 0;
        priority_queue<pii, vector<pii>, greater<pii>> pq;
        pq.push({0, start});

        while (!pq.empty()) {
            int u = pq.top().second;
            int dist = pq.top().first;
            pq.pop();

            if (dist > distances[u]) continue;

            for (const auto& edge : graph[u]) {
                int v = edge.first;
                int weight = edge.second;

                if (distances[u] + weight < distances[v]) {
                    distances[v] = distances[u] + weight;
                    pq.push({distances[v], v});
                }
            }
        }

        return distances;
    }

    int findTheCity(int n, vector<vector<int>>& edges, int distanceThreshold) {
        graph = vector<vector<pii>>(n);

        for (const auto& edge : edges) {
```

```
            int u = edge[0];
            int v = edge[1];
            int weight = edge[2];
            graph[u].emplace_back(v, weight);
            graph[v].emplace_back(u, weight);
        }

        int minReachableCount = INT_MAX;
        int resultCity = -1;

        for (int i = 0; i < n; ++i) {
            vector<int> distances = dijkstra(n, i, distanceThreshold);
            int reachableCount = 0;

            for (int j = 0; j < n; ++j) {
                if (i != j && distances[j] <= distanceThreshold) {
                    reachableCount++;
                }
            }

            if (reachableCount < minReachableCount || (reachableCount ==
minReachableCount && i > resultCity)) {
                minReachableCount = reachableCount;
                resultCity = i;
            }
        }

        return resultCity;
    }
};
```

Approach for Bellman-Ford algorithm:

To solve the problem of finding the city with the smallest number of reachable cities within a given distance threshold using the Bellman-Ford algorithm, follow these steps:

Step-by-Step Approach

1) **Graph Representation**:
    a) Use an edge list to store the graph. Each edge will be represented as a triplet (u, v, w) where u and v are the cities and w is the weight (distance).

2) **Bellman-Ford Algorithm Implementation**:
   a) Initialize a distance array to store the shortest distance from the starting city to each other city. Set all distances to infinity (`INT_MAX`), except the starting city, which is set to 0.
   b) Relax all edges |V| - 1 times (where |V| is the number of vertices).
   c) Optionally, a final pass through all edges can detect negative weight cycles, but this is unnecessary for this problem since distances are non-negative.
3) **Iterate and Count Reachable Cities**:
   a) For each city, run the Bellman-Ford algorithm to get the shortest distances to all other cities.
   b) Count how many cities have a distance less than or equal to the distance threshold.
4) **Track Minimum Reachable Cities**:
   a) Keep track of the minimum number of reachable cities and the corresponding city index. If there is a tie, choose the city with the larger index.

Solution Code for Bellman-Ford's algorithm:

```
#include <vector>
#include <limits.h>
using namespace std;

class Solution {
public:
    vector<int> bellmanFord(int n, int start, const vector<vector<int>>& edges) {
        vector<int> dist(n, INT_MAX);
        dist[start] = 0;

        for (int i = 0; i < n - 1; ++i) {
            for (const auto& edge : edges) {
                int u = edge[0];
                int v = edge[1];
                int weight = edge[2];

                if (dist[u] != INT_MAX && dist[u] + weight < dist[v]) {
                    dist[v] = dist[u] + weight;
                }
            }
        }

        return dist;
```

```
    }

    int findTheCity(int n, vector<vector<int>>& edges, int distanceThreshold) {
        int minReachableCount = INT_MAX;
        int resultCity = -1;

        for (int i = 0; i < n; ++i) {
            vector<int> distances = bellmanFord(n, i, edges);
            int reachableCount = 0;

            for (int j = 0; j < n; ++j) {
                if (i != j && distances[j] <= distanceThreshold) {
                    reachableCount++;
                }
            }

            if (reachableCount < minReachableCount || (reachableCount ==
minReachableCount && i > resultCity)) {
                minReachableCount = reachableCount;
                resultCity = i;
            }
        }

        return resultCity;
    }
};
```