ZJU-FDS-cx2020-Projects (Normal)

Dijkstra Sequence

3180103000 许乐乐

Date:2020-12-2

1. Introduction

The Shortest Path Problem is a classical algorithm problem in graph theory. It aims to find the shortest path between two nodes in a graph (composed of nodes and paths). The specific forms of the algorithm include:

- *Single Source Shortest Path Problem*:

  To find the shortest paths from one particular source vertex to all the other vertices of the given graph.

- *Multi Source Shortest Path Problem*:

  To find the shortest paths between any two vertices.

Dijkstra's algorithm is used for solving *Single Source Shortest Path Problem*. It was conceived by computer scientist Edsger W. Dijkstra in 1956 and published three years later.

Dijkstra's algorithm uses *Breadth First Search* to solve the single source shortest path problem of weighted digraph, and finally obtains a shortest path tree. This algorithm is often used in routing algorithms or as a sub module of other graph algorithms. For example, if the vertices in the graph represent cities and the weight on the edges represents the distance between cities, the algorithm can be used to find the shortest path between two cities.

The original *Dijkstra's algorithm* does not use the minimum priority queue, and the time complexity is $O(V^2)$, where V is the number of vertices of the graph. The time complexity of *Dijkstra's algorithm* implemented by Fibonacci heap is $O(E + VlogV)$, where E is the number of edges of the graph. It is the fastest algorithm for *Single Source Shortest Path Problem* ever known.

In *Dijkstra's algorithm*, a set contains vertices included in shortest path tree is maintained. During each step, we find one vertex which is not yet included and has a minimum distance from the source, and collect it into the set. Hence step by step an ordered sequence of vertices, let's call it *Dijkstra Sequence*, is generated by *Dijkstra's algorithm*. On the other hand, for a given graph, there could be more than one Dijkstra Sequence.

2. Algorithm Specification

We specify the algorithm used to check whether a given sequence is *Dijkstra Sequence* or not.

Algorithm: int main()

Input: Each input file contains one test case. For each case, the first line contains two positive integers $Nv(\leq 10^3)$ and $Ne(\leq 10^5)$, which are the total numbers of vertices and edges, respectively. Hence the vertices are numbered from 1 to Nv. Then Ne lines follow, each describes an edge by giving the indices of the vertices at the two ends, followed by a positive integer weight (≤ 100) of the edge. It is guaranteed that the given graph is connected. Finally the number of queries, K, is given as a positive integer no larger than 100, followed by K lines of sequences, each contains a permutation of the Nv vertices. It is assumed that the first vertex is the source for each sequence. All the inputs in a line are separated by a space.

Output: For each of the K sequences, print in a line `Yes` if it is a Dijkstra Sequence, or `No` if not.

Main Idea: Read in the related information of the given graph and stored in adjacency matrix(G). In each case, according to the input order, judge whether the distance of the input vertex is equal to the shortest path length currently. If no, print `No` ; if yes, print `Yes` .

Pseudo Code:

```
//Initialization(Nv,Ne,K,G,sequence,flag,visit,dist)
Read in the total number of vertices(Nv) and edges(Ne);
Initialize adjacency matrix(G), where each edge is initialized to infinity;
Read in and tore edge information in adjacency matrix(G);
Read in the number of queries(K);
while(K--){
    Read in the sequence to be judged(sequence);
    bool flag = true;
    Initialize the matrix(visit) to indicate whether the vertex has been 
accessed, where each vertex is initialized as not accessed;
    Initialize the matrix(dist) to indicate the shortest path length between one 
particular source vertex to all the other vertices of the given graph, where 
each path length is initialized to INF except that the source point is assigned 
0;
    //Begin checking
    for (int i = 1; i <= Nv; i++) {
        int minN = -1, minDist = INF;
        Find the minimum path length(minDist) and the corresponding vertex(minN) 
in the currently non-visited vertices;
        if (minN == -1 || dist[minN] != dist[sequence[i]]){
            flag=false;
            continue;
        }
        For each of its adjacent vertix,if the shortest path length becomes 
shorter after adding minN, update it.
        if (flag) printf("Yes\n");
        else printf("No\n");
    }
}
```

3. Testing Results

Table 1 shows some typical test cases for verifying the *Dijkstra sequence* implementation and capturing potential bugs.

Test Cases	Design Purpose	result	status
5 7 1 2 2 1 5 1 2 3 1 2 4 1 2 5 2 3 5 1 3 4 1 7 5 1 3 4 2(a.) 5 3 1 2 4(b.) 2 3 4 5 1(c.) 3 2 1 5 4(d.) 3 2 5 4 1(e.) 3 2 5 1 4(f.) 3 2 1 5 4(g.)	(a.)(b.)(c.)(e.) are all in correct order. (d.)(g.) the 3rd should be 4 or 5. (f.) the 4th should be 4.	Yes Yes Yes No Yes No No	*pass*
data in "TestData.txt" in folder "code"	large size, $N_v = 50 \& N_e = 100$	Yes Yes Yes Yes No	*pass*
2 1 1 2 1 2 1 2 2 1	boundary example $N_v = 2 \& N_e = 1$	Yes Yes	*pass*
3 1 1 2 1 1 1 2 3	the graph without shortest path, which has isolated vertices.	No	*pass*

Table 1 : Test cases for the *Dijkstra Sequence* implementation.

4. Analysis and Comments

For Initialization, the time complexity is $O({N_v}^2)$. For each case to be checked, there are two subtasks inside the for-loop.

Specifially,

- Find the minimum path length(minDist) and the corresponding vertex(minN) in the currently non-visited vertices;
- For each of its adjacent vertix, if the shortest path length becomes shorter after adding minN, update it.

This is independent of any particular input. Therefore, the time complexity is $O({N_v}^2)$. Because there are k queries, the time complexity should be multiplied by k. So the total time complexity is $O(k * {N_v}^2)$.

There is still considerable room for improvement. For example, we could use Fibonacci heap and adjacency list to achieve an $O(k * (N_e + N_V logN_v))$ algorithm, which is better for sparse graph.

For the space requirement, since we use adjacency matrix to store, the space complexity is $O({N_v}^2)$, which is better for dense graph. If the graph is sparse, we can use adjacency list to store, and the space complexity is $O(N_v + N_e)$.

Appendix: Source Code (in C)

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <stdbool.h>
#define MaxNv 2000
#define INF 1e9

int dist[MaxNv], sequence[MaxNv], G[MaxNv][MaxNv];
bool visit[MaxNv];

int main() {
    int Nv, Ne, K, x, y, z;

    //Enter the total number of vertices(Nv) and edges(Ne).
    scanf("%d %d", &Nv, &Ne);

    //Initialize adjacency matrix(G).
    for (int i = 0; i < MaxNv; i++) {
        for (int j = 0; j < MaxNv; j++) {
            G[i][j] = INF;  //Each edge is initialized to infinity.
        }
    }

    //Store edge information in adjacency matrix(G).
    for (int i = 0; i < Ne; i++) {
        scanf("%d %d %d", &x, &y, &z);
        G[x][y] = G[y][x] = z;
    }

    //Enter the number of queries(K).
    scanf("%d", &K);

    //Judge each case.
    while (K--) {

        //Enter the sequence to be judged.
        for (int i = 1; i <= Nv; i++) {
            scanf("%d", &sequence[i]);
        }

        //flag to judge whether it is a Dijkstra sequence.
        bool flag = true;

        //Initialize the matrix(visit) to indicate whether the vertex has been
accessed.
        for (int i = 0; i < MaxNv; i++) {
            visit[i] = false;  //Each vertex is initialized as not accessed.
        }

        //Initialize the matrix(dist) to indicate the shortest
        //path length between one particular source vertex to
        //all the other vertices of the given graph.
        for (int i = 0; i < MaxNv; i++) {
            dist[i] = INF;  //Each path length is initialized to INF.
        }

        //The shortest path length from oneself to oneself is initialized to 0.
        dist[sequence[1]] = 0;

        /*
        * According to the input order,judge whether the distance
        * of the input vertex is the shortest path length                 *
currently.If no,return false;if yes,update the matrix(dist).
        */
        for (int i = 1; i <= Nv; i++) {
            int minN = -1, minDist = INF;

            /*
            * Find the minimum path length(minDist) and the corresponding
vertex(minN)
            * in the currently non-visited vertices.
            */
            for (int j = 1; j <= Nv; j++) {
                if (!visit[j] && dist[j] < minDist) {
                    minN = j;
                    minDist = dist[j];
                }
            }

            /*
            * If the minimum path is not found or the distance
            * of the input vertex is not equal to the current
            * shortest path length,then failed.
            */
            if (minN == -1 || dist[minN] != dist[sequence[i]]) {
                flag = false;
                continue;
            }

            //Update the information.
            minN = sequence[i];
            visit[minN] = true;

            /*
            * For each of its adjacent vertix,if the shortest
            * pathlength becomes shorter after adding minN,                  *
update it.
            */
            for (int j = 1; j <= Nv; j++) {
                if (!visit[j] && G[minN][j] != INF && G[minN][j] + dist[minN] <
dist[j]) {
                    dist[j] = G[minN][j] + dist[minN];
                }
            }
        }

        //Judge according to the flag.
        if (flag) printf("Yes\n");
        else printf("No\n");
    }
    getchar();
    getchar();
}
```

Declaration

I hereby declare that all the work done in this project is of my independent effort.