November 23, 2022 ▪ Data Structure / Graph

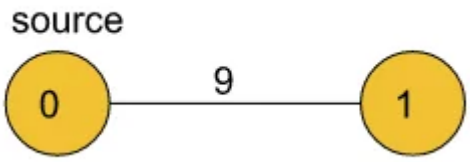
Dijkstra's Algorithm – Using Priority Queue : G-32

Given a weighted, undirected, and connected graph of **V** vertices and an adjacency list adj where adj[i] is a list of lists containing two integers where the **first** integer of each list j denotes there is an **edge** between i and j, second integers corresponds to the **weight** of that edge. You are given the source vertex **S** and You have to Find the shortest distance of all the vertex from the source vertex **S**. You have to return a list of integers denoting the shortest distance between **each node** and the Source vertex **S**.

Note: The Graph doesn't contain any negative weight cycle.

Examples:

Example 1:



Input:

V = 2

adj [] = {{{1, 9}}, {{0, 9}}}

S = 0

Output:

0 9

Explanation:

The source vertex is 0. Hence, the shortest distance of node 0 from the source is 0 and the shortest distance of node 1 from source will be 9.

Example 2:

Input:

Count Subarray sum Equals K

Binary Tree Representation in Java

Accolite Digital Amazon Arcesium arrays Bank of America Barclays BFS Binary Search Binary Search Tree Commvault CPP DE Shaw DFS DSA Self Paced google HackerEarth Hashing infosys inorder Interview Experience Java Juspay Kreeti Technologies Morgan Stanley Newfold Digital Oracle post order recursion Samsung SDE Core Sheet SDE Sheet Searching set-bits sorting Strivers A2ZDSA Course sub-array subarray Swiggy takeuforward TCS TCS CODEVITA TCS Ninja

V = 3, E = 3

adj = {{{1, 1}, {2, 6}}, {{2, 3}, {0, 1}}, {{1, 3}, {0, 6}}}

S = 2

Output:

4 3 0

Explanation:

For nodes 2 to 0, we can follow the path 2-1-0. This has a distance of 1+3 = 4, whereas the path 2-0 has a distance of 6. So, the Shortest path from 2 to 0 is 4.

The shortest distance from 0 to 1 is 1.

Solution

Disclaimer: Don't jump directly to the solution, try it out yourself first.

Problem Link

Note: In case any image/dry run is not clear please refer to the video attached at the bottom.

Approach:

We'll be using Priority Queue in this approach for finding the shortest distances from the source node to every other node through Dijkstra's Algorithm.

Initial configuration:

- **Source Node:** Before starting off with the Algorithm, we need to define a source node from which the shortest distances to every other node would be calculated.
- **Priority Queue:** Define a Priority Queue which would contain pairs of the type {dist, node}, where 'dist' indicates the currently updated value of the shortest distance from the source to the 'node'.
- **Dist Array:** Define a dist array initialized with a large integer number at the start indicating that the nodes are unvisited initially. This array stores the shortest distances to all the nodes from the source node.

The Algorithm consists of the following steps :

- We would be using a min-heap and a distance array of size V (where 'V' are the number of nodes in the graph) initialized with infinity (indicating that at present none of the nodes are reachable from the source node) and initialize the distance to source node as 0.
- We push the source node to the queue along with its distance which is also 0.
- For every node at the top of the queue, we pop the element out and look out for its adjacent nodes. If the current reachable

distance is better than the previous distance indicated by the distance array, we update the distance and push it into the queue.

A node with a lower distance would be at the top of the priority queue as opposed to a node with a higher distance because we are using a min-heap. By following step 3, until our queue becomes empty, we would get the minimum distance from the source node to all other nodes. We can then return the distance array. Here’s a quick demonstration of the algorithm :

Note: Dijkstra’s Algorithm is not valid for negative weights or negative cycles.

We can understand that by looking at the illustration below :

Here, we initially mark the source node '0' as 0 and node '1' as INFINITY (as it is unvisited). Now, when we start applying the above algorithm on this we notice that both the nodes are updated each time we come to them again. This is due to the negative weight of '-2' which makes the total distance to the node always lesser than the previous value. Therefore, due to inaccurate results, we assume the graph to always contain positive weights while using Dijkstra's Algorithm.

Note: If you wish to see the dry run of the above approach, you can watch the video attached to this article.

Intuition:

The above problem seems familiar to finding the shortest distance in the case of unit edge weights for undirected graphs. Hence, our first guess could be a BFS kind of approach. The only thing that we need to take care of is

that for all the paths possible between a pair of nodes, we need to store the path with the minimum cost between them. That is, say we have a node that has been reached by two paths, one with a cost of 7 and another with a cost of say 9. It is obvious that the path with a cost of 7 would be more optimal than the path with a cost of 9.

For knowing the intuition behind why Priority Queue is being used for storing distances of nodes from the source, please watch this video of the series to get a clear understanding of why instead a Queue is not being used in this approach.

Code:

C++ Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution
{
public:
    // Function to find the shortest
    // from the source vertex S.
    vector<int> dijkstra(int V, vect
    {

        // Create a priority queue f
        // where dist is the distanc
        priority_queue<pair<int, int

        // Initialising distTo list
        // indicate the nodes are un
        // This list contains distan
        vector<int> distTo(V, INT_MA
```

```
        // Source initialised with d
        distTo[S] = 0;
        pq.push({0, S});

        // Now, pop the minimum dist
        // and traverse for all its
        while (!pq.empty())
        {
            int node = pq.top().seco
            int dis = pq.top().first
            pq.pop();

            // Check for all adjacen
            // element whether the p
            for (auto it : adj[node]
            {
                int v = it[0];
                int w = it[1];
                if (dis + w < distTo
                {
                    distTo[v] = dis

                    // If current di
                    // push it into
                    pq.push({dis + w
                }
            }
        }
        // Return the list containin
        // from source to all the no
        return distTo;
    }
};

int main()
{
    // Driver code.
    int V = 3, E = 3, S = 2;
    vector<vector<int>> adj[V];
    vector<vector<int>> edges;
    vector<int> v1{1, 1}, v2{2, 6},
    int i = 0;
    adj[0].push_back(v1);
    adj[0].push_back(v2);
    adj[1].push_back(v3);
    adj[1].push_back(v4);
```

```
    adj[2].push_back(v5);
    adj[2].push_back(v6);

    Solution obj;
    vector<int> res = obj.dijkstra(V

    for (int i = 0; i < V; i++)
    {
        cout << res[i] << " ";
    }
    cout << endl;
    return 0;
}
```

Output:

4 3 0

Time Complexity: O(E log(V)), Where E = Number of edges and V = Number of Nodes.

Space Complexity: O(|E| + |V|), Where E = Number of edges and V = Number of Nodes.

Java Code

```
import java.util.*;
import java.lang.*;
import java.io.*;

public class Main {

    public static void main(String[]
        int V = 3, E=3,S=2;
    ArrayList<Integer> node1 = new A
    ArrayList<Integer> node2 = new A
    ArrayList<Integer> node3 = new A
    ArrayList<Integer> node4 = new A
    ArrayList<Integer> node5 = new A
    ArrayList<Integer> node6 = new A

    ArrayList<ArrayList<Integer>> in
      {
```

```
                add(node1);
                add(node2);
            }
        };
        ArrayList<ArrayList<Integer>> in
            {
                add(node3);
                add(node4);
            }
        };
        ArrayList<ArrayList<Integer>> in
            {
                add(node5);
                add(node6);
            }
        };
        ArrayList<ArrayList<ArrayList<In
            {
                add(inter1); // for 1st
                add(inter2); // for 2nd
                add(inter3); // for 3rd
            }
        };
        //add final values of adj here.
        Solution obj = new Solution();
        int[] res= obj.dijkstra(V,adj,S)

        for(int i=0;i<V;i++){
            System.out.print(res[i]+" ")
        }
        System.out.println();

        }
    }

    class Pair{
        int node;
        int distance;
        public Pair(int distance,int nod
            this.node = node;
            this.distance = distance;
        }
    }
    //User function Template for Java
    class Solution
    {
```

```
        //Function to find the shortest
        //from the source vertex S.
        static int[] dijkstra(int V, Arr
        {
            // Create a priority queue f
            // where dist is the distanc
            PriorityQueue<Pair> pq =
            new PriorityQueue<Pair>((x,y

            int []dist = new int[V];

            // Initialising distTo list
            // indicate the nodes are un
            // This list contains distan
            for(int i = 0;i<V;i++) dist[

            // Source initialised with d
            dist[S] = 0;
            pq.add(new Pair(0,S));

            // Now, pop the minimum dist
            // and traverse for all its
            while(pq.size() != 0) {
                int dis = pq.peek().dist
                int node = pq.peek().nod
                pq.remove();

                // Check for all adjacen
                // element whether the p
                for(int i = 0;i<adj.get(
                    int edgeWeight = adj
                    int adjNode = adj.ge

                    // If current distan
                    // push it into the
                    if(dis + edgeWeight
                        dist[adjNode] =
                        pq.add(new Pair(
                    }
                }
            }
            // Return the list containin
            // from source to all the no
            return dist;
        }
    }
```

Output:

4 3 0

Time Complexity: O(E log(V)), Where E = Number of edges and V = Number of Nodes.

Space Complexity: O(|E| + |V|), Where E = Number of edges and V = Number of Nodes.

> Special thanks to **Priyanshi Goel** for contributing to this article on takeUforward. If you also wish to share your knowledge with the takeUforward fam, please check out this article. If you want to suggest any improvement/correction in this article please mail us at write4tuf@gmail.com

« Previous Post
Word Ladder-II (Optimised Approach) G-31

Next Post »
Dijkstra's Algorithm – Using Set : G-33

Load Comments