November 23, 2022 ▪ Data Structure / Graph

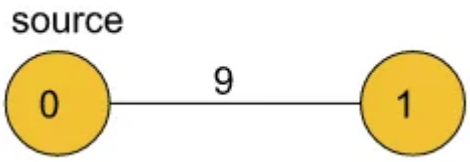
Dijkstra's Algorithm – Using Set : G-33

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 Source vertex **S**.

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

Example 1:

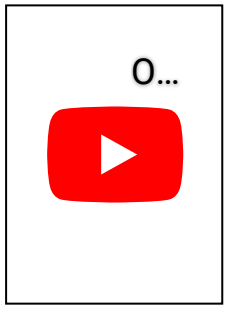
```
Input :
```




```
V = 2
adj [] = {{{1, 9}}, {{0, 9}}}
S = 0

Output:
0 9

Explanation: :
The source vertex is 0. Hence, the s
from the source is 0 and the shortes
source will be 9.
```

Example 2:

```
Input:

V = 3, E = 3
adj = {{{1, 1}, {2, 6}}, {{2, 3}, {0
S = 2
Output:
4 3 0
```



Recent Posts

Implement Upper Bound

Implement Lower Bound

2023 – Striver's SDE Sheet Challenge

Top Array Interview Questions – Structured Path with Video Solutions

Longest Subarray with sum K | [Postives and Negatives]

```
Explanation:
For nodes 2 to 0, we can follow the
This has a distance of 1+3 = 4, whe
has a distance of 6. So, the Shorte

The shortest distance from 0 to 1 i
```

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 Set 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.
- **Set:** Define a Set that 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 set 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 set along with its distance which is also 0.
- Now, we start erasing the elements from the set and look out for their adjacent nodes one by one. If the current reachable distance is better than the previous distance indicated by the distance array, we update the distance and insert it in the set.
- A node with a lower distance would be first erased from the set as opposed to a node with a higher distance. By following step 3, until our set becomes empty, we would get the minimum distance from the

source node to all other nodes. We can then return the distance array.

- The only difference between using a Priority Queue and a Set is that in a set we can check if there exists a pair with the same node but a greater distance than the current inserted node as there will be no point in keeping that node into the set if we come across a much better value than that. So, we simply delete the element with a greater distance value for the same node.
- 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 implements a BFS kind of approach using the set data structure. 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. A set data structure in C++ always stores the elements in increasing order i.e., when we erase from a set, the smallest valued elements get erased first.

Code:

C++ Code

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

class Solution
{
    public:
      //Function to find the shortes
    //from the source vertex S.
    vector <int> dijkstra(int V, vec
    {
        // Create a set ds for stori
        // where dist is the distanc
        // set stores the nodes in a
        set<pair<int,int>> st;

        // Initialising dist list wi
        // indicate the nodes are un
        // This list contains distan
        vector<int> dist(V, 1e9);

        st.insert({0, S});

        // Source initialised with d
        dist[S] = 0;

        // Now, erase the minimum di
        // and traverse for all its
        while(!st.empty()) {
```

```
                auto it = *(st.begin());
                int node = it.second;
                int dis = it.first;
                st.erase(it);

                // Check for all adjacen
                // element whether the p
                for(auto it : adj[node])
                    int adjNode = it[0];
                    int edgW = it[1];

                    if(dis + edgW < dist
                        // erase if it w
                        // a greater cos
                        if(dist[adjNode]
                            st.erase({di

                        // If current di
                        // push it into
                        dist[adjNode] =
                        st.insert({dist[
                     }
                }
            }
            // Return the list containin
            // from source to all the no
            return dist;
        }
};

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

Note: This set approach cannot be implemented in **JAVA** by using TreeSet or HashSet. For implementing Dijkstra's Algorithm in **JAVA**, we would use a priority queue only.

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

Next Post »
Bellman Ford Algorithm: G-41

Load Comments

takeUforward

The best place to learn data structures, algorithms, most asked coding interview questions, real interview experiences free of cost.

Follow Us

DSA Playlist	DSA Sheets	Contribute
Array Series	Striver's SDE Sheet	Write an Article
Tree Series	Striver's A2Z DSA Sheet	
Graph Series	SDE Core Sheet	
DP Series	Striver's CP Sheet	