takeUforward

Search

December 20, 2022   ▪   Data Structure / Graph

G-35 : Print Shortest Path – Dijkstra's Algorithm

Problem Statement:

You are given a weighted undirected graph having **n+1** vertices numbered from 0 to n and **m** edges describing there are edges between a to b with some weight, find the shortest path between the vertex 1 and the vertex n, and if the path does not exist then return a list consisting of only **-1**.

Note: Please read the G-32 and the G-33 article before reading this article to get a clear understanding of Dijkstra's Algorithm will form the base for this particular problem.
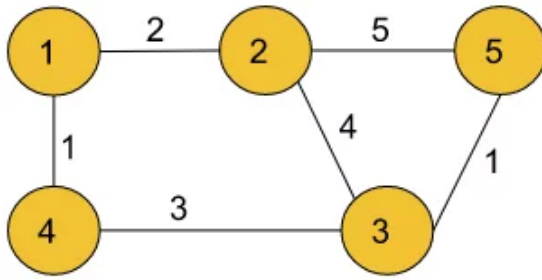
Examples:

Example 1:

Latest Video on takeUforward

Latest Video on Striver

Input:
n = 5, m= 6
edges = [[1,2,2], [2,5,5], [2,3,4],
Output:
1 4 3 5
Explanation:
The source vertex is 1. Hence, the
of node 5 from the source will be 1
the path with a minimum sum of edge
to destination.

Example 2:

Input:
V = 4, E = 4
edges = [[1,2,2], [2,3,4], [1,4,1],
Output: 1 4
Explanation:

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]

```
The source vertex is 1. Hence, the
path of node 4 from the source will
the path with the minimum sum of ed
source to destination.
```

Solution

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

[Problem Link](#)

Approach:

We'll be using Dijkstra's Algorithm with a slight modification for solving this particular problem.

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.

- **Parent Array:** This array's sole purpose is
  to store the parent of a particular node
  i.e., the node from where that node came
  while traversing through the graph by
  Dijkstra's Algorithm.

The Algorithm consists of the following steps
:

- We start by initializing an adjacency list
  which will store all the adjacent nodes for
  a particular node along with the weights
  associated with them.
- Then, as a part of the initial configuration,
  we define a dist array to store the
  updated shortest distances for each node,
  a priority queue for storing the distance-
  node pairs, and a source node.
- In addition to this, we also declare a
  'parent' array which would store the
  parent node for each node and will update
  itself to a different parent if a shorter path
  from a node is found at some point in
  time.
- At the start, all nodes' parents have been
  set to the nodes themselves to indicate
  that the traversal has not yet been
  started.
- 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 (dis + edW < dist[adjNode]),
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.

- In addition to the previous step, we will
  also update the parent array to the node
  from where the current node came while
  traversing.

- By following step 5 repeatedly until our
  queue becomes empty, we would get the
  minimum distance from the source node
  to all other nodes and also our parent
  array would be updated according to the
  shortest path.

- Now, we run a loop starting from the
  destination node storing the node's
  parent and then moving to the parent
  again (backtrack) till the parent[node]
  becomes equal to the node itself.

- At last, we reverse the array in which the
  path is being stored as the path is in
  reverse order. Finally, we return the 'path'
  array.

- Here's a quick demonstration of the
  algorithm :

In a similar way, we can perform the iterations for all the other nodes and hence return the path.

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

Intuition:

The intuition behind the above problem is based on Dijkstra's Algorithm with a combination of a little bit of memoization in order to **print** the shortest possible path and not just calculate the shortest distance between the source and the destination node. In order to print the path we will try to remember the node from which we came while traversing each node by Dijkstra's Algorithm along with calculating the shortest distance.

An array called 'parent' can be used for this purpose which would store the parent node for each node and will update itself if a shorter path from a node is found at some point in time. This will help us to print the path easily at the end by backtracking through the parent array till we reach the source node.

Code:

C++ Code

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

class Solution
{
public:
    vector<int> shortestPath(int n,
    {
        // Create an adjacency list
        // where the edge weight is
        vector<pair<int, int>> adj[n
        for (auto it : edges)
        {
            adj[it[0]].push_back({it
            adj[it[1]].push_back({it
        }
        // Create a priority queue f
        // in the form of a pair { d
        priority_queue<pair<int, int

        // Create a dist array for s
        //for storing the nodes from
        // the parent array came fro
        vector<int> dist(n + 1, 1e9)
        for (int i = 1; i <= n; i++)
                parent[i] = i;

        dist[1] = 0;
```

```cpp
            // Push the source node to t
            pq.push({0, 1});
            while (!pq.empty())
            {
                // Topmost element of th
                auto it = pq.top();
                pq.pop();
                int node = it.second;
                int dis = it.first;

                // Iterate through the a
                for (auto it : adj[node]
                {
                    int adjNode = it.fir
                    int edW = it.second;

                    // Check if the prev
                    // greater than the
                    // if yes then updat
                    if (dis + edW < dist
                    {
                        dist[adjNode] =
                        pq.push({dis + e

                        // Update the pa
                        // node where it
                        parent[adjNode]
                    }
                }
            }

            // If distance to a node cou
            if (dist[n] == 1e9)
                return {-1};

            // Store the final path in t
            vector<int> path;
            int node = n;

            // Iterate backwards from de
            while (parent[node] != node)
            {
                path.push_back(node);
                node = parent[node];
            }
            path.push_back(1);
```

```cpp
            // Since the path stored is
            // to get the final answer a
            reverse(path.begin(), path.e
            return path;
        }
    };

    int main()
    {
        // Driver Code

        int V = 5, E = 6;
        vector<vector<int>> edges = {{1,
        {3, 5, 1}};
        Solution obj;
        vector<int> path = obj.shortestP

        for (int i = 0; i < path.size();
        {

            cout << path[i] << " ";
        }
        cout << endl;
        return 0;
    }
```

Output :

1 4 3 5

Time Complexity: O(E log(V)) { for Dijkstra's
Algorithm } + O(V) { for backtracking in order
to find the parent for each node } Where E =
Number of edges and V = Number of Nodes.

Space Complexity: O(|E| + |V|) { for priority
queue and dist array } + O(|V|) { for storing
the final path } Where E = Number of edges
and V = Number of Nodes.

Java Code ▾

```java
import java.util.*;

class Pair{
    int first;
    int second;
    public Pair(int first,int second
        this.first = first;
        this.second = second;
    }
}
class Solution {
    public static List<Integer> shor

        // Create an adjacency list
        // where the edge weight is
        ArrayList<ArrayList<Pair>> a
        for(int i = 0;i<=n;i++) {
            adj.add(new ArrayList<>(
        }
        for(int i = 0;i<m;i++) {
            adj.get(edges[i][0]).add
            adj.get(edges[i][1]).add
        }

        // Create a priority queue f
        // in the form of a pair { d
        PriorityQueue<Pair> pq =
        new PriorityQueue<Pair>((x,y

        // Create a dist array for s
        //for storing the nodes from
        // the parent array came fro
        int[] dist = new int[n+1];
        int[] parent =new int[n+1];
        for(int i = 1;i<=n;i++) {
            dist[i] = (int)(1e9);
            parent[i] = i;
        }

        dist[1] = 0;

        // Push the source node to t
        pq.add(new Pair(0, 1));
```

```java
            while(pq.size() != 0) {

                // Topmost element of th
                Pair it = pq.peek();
                int node = it.second;
                int dis = it.first;
                pq.remove();

                // Iterate through the a
                for(Pair iter : adj.get(
                    int adjNode = iter.f
                    int edW = iter.secon

                    // Check if the prev
                    // greater than the
                    // if yes then updat
                    if(dis + edW < dist[
                        dist[adjNode] =
                        pq.add(new Pair(

                        // Update the pa
                        // node where it
                        parent[adjNode]
                    }
                }
            }

            // Store the final path in t
            List<Integer> path = new Arr

            // If distance to a node cou
            if(dist[n] == 1e9) {
                path.add(-1);
                return path;
            }

            int node = n;
            // o(N)
            while(parent[node] != node)
                path.add(node);
                node = parent[node];
            }
            path.add(1);

            // Since the path stored is
            // to get the final answer a
```

```
            Collections.reverse(path);
            return path;
        }
    }

    class tuf {

        public static void main(String[]
            int V = 5, E = 6;

            int[][] edges = {{1,2,2},{2,

            Solution obj = new Solution(
            List < Integer > path = obj.
            for (int i = 0; i < path.siz
                System.out.print(path.ge
            }
            System.out.println();
        }
    }
```

Output :

1 4 3 5

Time Complexity: O(E log(V)) { for Dijkstra's
Algorithm } + O(V) { for backtracking in order
to find the parent for each node } Where E =
Number of edges and V = Number of Nodes.

Space Complexity: O(|E| + |V|) { for priority
queue and dist array } + O(|V|) { for storing
the final path } 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   Next Post »

Kruskal's Algorithm   **G-34: Dijkstra's**
– Minimum   **Algorithm (Intuition**
Spanning Tree : G-47   **and Time Complexity**
                               Derivation)

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