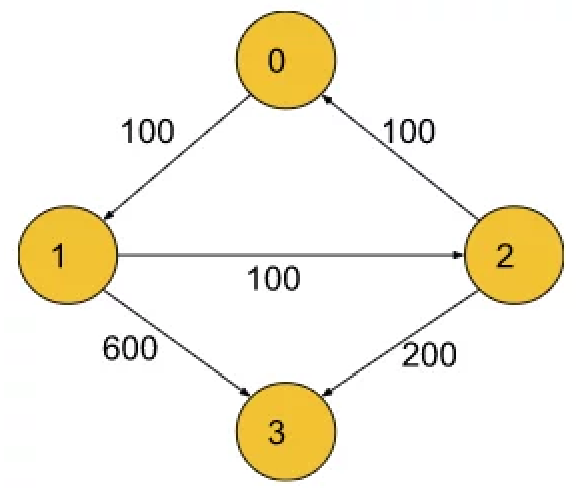
takeUforward

January 10, 2023 ▪ Data Structure / Graph

G-38: Cheapest Flights Within K Stops

There are n cities and m edges connected by some number of flights. You are given an array of flights where flights[i] = [$from_i$, to_i, $price_i$] indicates that there is a flight from city $from_i$ to city to_i with cost price. You have also given three integers src, dst, and k, **and return *the cheapest price*** *from* src *to* dst *with at most* k *stops.* If there is no such route, return -1.

Example 1:




```
Input:
n = 4
flights = [[0,1,100],[1,2,100],[2,0,
src = 0
dst = 3
k = 1
Output:
700
Explanation:
The optimal path with at most 1 sto
Note that the path through cities [(
```

Example 2:

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]

```
Input:
n = 3
flights = [[0,1,100],[1,2,100],[0,2,
src = 0
dst = 2
k = 1
Output:
200
Explanation:
The graph is shown above.
The optimal path with at most 1 sto
```

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.

Intuition:

Since in this problem, we have to calculate the minimum cost to reach the destination from the source but with a restriction on the number of stops, we would be using Dijkstra's Algorithm. Now, one must wonder that based on what parameter we should add elements to the priority queue.

Now, if we store the elements in the priority queue with the priority given to the minimum distance first, then after a few iterations we would realize that the Algorithm will halt when the number of stops would exceed. This may result in a wrong answer as it would not allow us to explore those paths which have more cost but fewer stops than the current answer.

To tackle this issue, we store the elements in terms of the minimum number of **stops** in the priority queue so that when the algorithm halts, we can get the minimum cost within limits.

Also, a point to note here is that do we really need a priority queue for carrying out the algorithm? The answer for that is **No** because when we are storing everything in terms of a number of stops, the stops are increasing monotonically which means that the number of sops is increasing by 1 and when we pop an element out of the queue, we are always popping the element with a lesser number of stops first. Replacing the priority queue with a simple queue will let us eliminate an extra log(N) of the complexity of insertion-deletion in a priority queue which would in turn make our algorithm a lot faster.

Approach:

Initial configuration:

- **Queue:** Define a Queue that would contain pairs of the type {stops, {node,dist} }, where 'dist' indicates the currently updated value of the distance from the source to the 'node' and 'stops' contains the number of nodes one has to traverse in order to reach node from src.

- **Distance Array:** Define a distance array that would contain the minimum cost/distance from the source cell to a particular cell. If a cell is marked as 'infinity' then it is treated as unreachable/unvisited.
- **Source and Destination:** Define the source and the destination from where the flights have to run.

The Algorithm consists of the following steps :

- Start by creating an adjacency list, a queue that stores the distance-node and stops pairs in the form {stops,{node,dist}} and a dist array with each node initialized with a very large number (to indicate that they're unvisited initially) and the source node marked as '0'.
- We push the source cell to the queue along with its distance which is also 0 and the stops are marked as '0' initially because we've just started.
- Pop the element at the front of the queue and look out for its adjacent nodes.
- If the current dist value of a node is better than the previous distance indicated by the distance array and the number of

stops until now is less than K, we update the distance in the array and push it to the queue. Also, increase the stop count by 1.

- We repeat the above three steps until the queue becomes empty. Note that we **do not** stop the algorithm from just reaching the destination node as it may give incorrect results.
- Return the calculated distance/cost after we reach the required number of stops. If the queue becomes empty and we don't encounter the destination node, return '-1' indicating there's no path from source to destination.

Here's a quick demonstration of the Algorithm's 1st iteration for **example 1** stated above (all the further iterations would be done in a similar way) :

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

attached to this article.

Code:

C++ Code

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

class Solution
{
public:
    int CheapestFLight(int n, vector
                       int src, int
    {
        // Create the adjacency list
        // the form of a graph.
        vector<pair<int, int>> adj[n
        for (auto it : flights)
        {
            adj[it[0]].push_back({it
        }

        // Create a queue which stor
        // source in the form of {st
        // the no. of nodes between
        queue<pair<int, pair<int, in

        q.push({0, {src, 0}});

        // Distance array to store t
        vector<int> dist(n, 1e9);
        dist[src] = 0;

        // Iterate through the graph
        // popping out the element w
        while (!q.empty())
        {
            auto it = q.front();
            q.pop();
            int stops = it.first;
            int node = it.second.fir
            int cost = it.second.sec

            // We stop the process a
```

```
            if (stops > K)
                continue;
            for (auto iter : adj[nod
            {
                int adjNode = iter.f
                int edW = iter.secon

                // We only update th
                // less than the pre
                if (cost + edW < dis
                {
                    dist[adjNode] =
                    q.push({stops +
                }
            }
        }
        // If the destination node i
        // else return the calculate
        if (dist[dst] == 1e9)
            return -1;
        return dist[dst];
    }
};

int main()
{
    // Driver Code.
    int n = 4, src = 0, dst = 3, K =

    vector<vector<int>> flights = {{
    {2, 3, 200}};

    Solution obj;

    int ans = obj.CheapestFLight(n,

    cout << ans;
    cout << endl;

    return 0;
}
```

Output :

700

Time Complexity: O(N) { Additional log(N) of time eliminated here because we're using a simple **queue** rather than a **priority queue** which is usually used in Dijkstra's Algorithm }.

Where N = Number of flights / Number of edges.

Space Complexity: O(|E| + |V|) { for the adjacency list, priority queue, and the dist array }.

Where E = Number of edges (flights.size()) and V = Number of Airports.

Java Code

```
import java.util.*;

class Pair{
    int first;
    int second;
    public Pair(int first,int second
        this.first = first;
        this.second = second;
    }
}
class Tuple {
    int first, second, third;
    Tuple(int first, int second, int
        this.first = first;
        this.second = second;
        this.third = third;
    }
}
class Solution {
    public int CheapestFLight(int n,
```

```
// Create the adjacency list
// the form of a graph.
ArrayList<ArrayList<Pair>> a
for(int i = 0;i<n;i++) {
    adj.add(new ArrayList<>(
}
int m = flights.length;
for(int i = 0;i<m;i++) {
    adj.get(flights[i][0]).a
}

// Create a queue which stor
// source in the form of {st
// the no. of nodes between
Queue<Tuple> q = new LinkedL

q.add(new Tuple(0, src, 0));

// Distance array to store t
int[] dist = new int[n];
for(int i = 0;i<n;i++) {
    dist[i] = (int)(1e9);
}
dist[src] = 0;

// Iterate through the graph
// popping out the element w
while(!q.isEmpty()) {
    Tuple it = q.peek();
    q.remove();
    int stops = it.first;
    int node = it.second;
    int cost = it.third;

    // We stop the process a
    if(stops > K) continue;
    for(Pair iter: adj.get(n
        int adjNode = iter.f
        int edW = iter.secon

        // We only update th
        // less than the pre
        if (cost + edW < dis
            dist[adjNode] =
            q.add(new Tuple(
        }
```

```
                }
            }
            // If the destination node i
            // else return the calculate
            if(dist[dst] == (int)(1e9))
            return dist[dst];
        }
    }

    class tuf {

        public static void main(String[]

            int n = 4, src = 0, dst = 3,
            int[][] flights={{0, 1, 100}

            Solution obj = new Solution(
            int ans = obj.CheapestFLight

            System.out.print(ans);
            System.out.println();
        }
    }
```

Output :

700

Time Complexity: O(N) { Additional log(N) of time eliminated here because we're using a simple **queue** rather than a **priority queue** which is usually used in Dijkstra's Algorithm }.

Where N = Number of flights / Number of edges.

Space Complexity: O(|E| + |V|) { for the adjacency list, priority queue, and the dist array }.

Where E = Number of edges (flights.size())

and V = Number of Airports.

> 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
G-37: Path With Minimum Effort

Next Post »
G-39: Minimum Multiplications to Reach End

Load Comments

DSA Playlist

Array Series

Tree Series

Graph Series

DP Series

DSA Sheets

Striver’s SDE Sheet

Striver’s A2Z DSA Sheet

SDE Core Sheet

Striver’s CP Sheet

Contribute

Write an Article