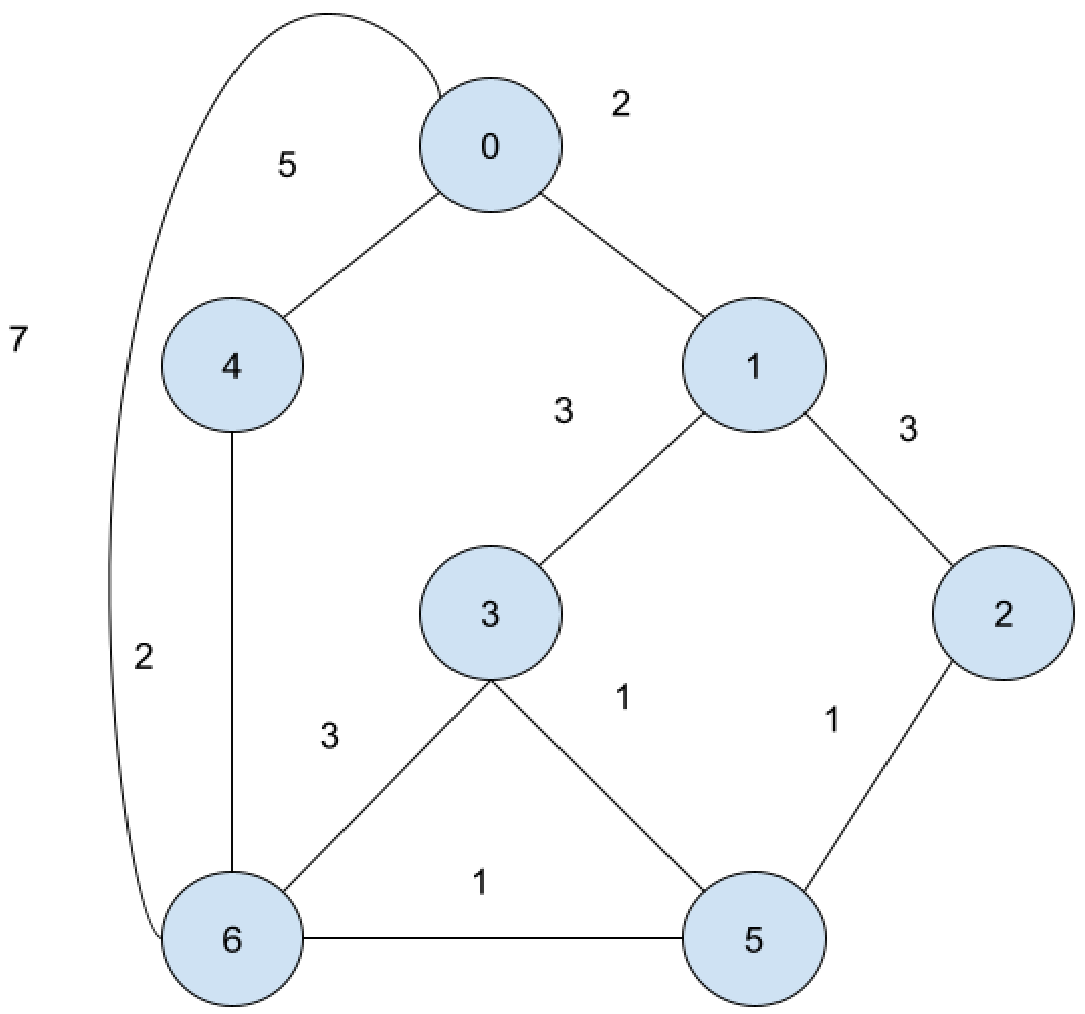
MEDIUM

Number of ways to arrive at destination

Intuition

Given : City of n intersection from 0-n-1
Bi-driectional road ie undirected graph
2D integer array roads roads[i] = [ui, vi, time] ie. from to and time taken to reach from u to v
We have to reach 0 to n-1 in shortest amount of time

Eg.



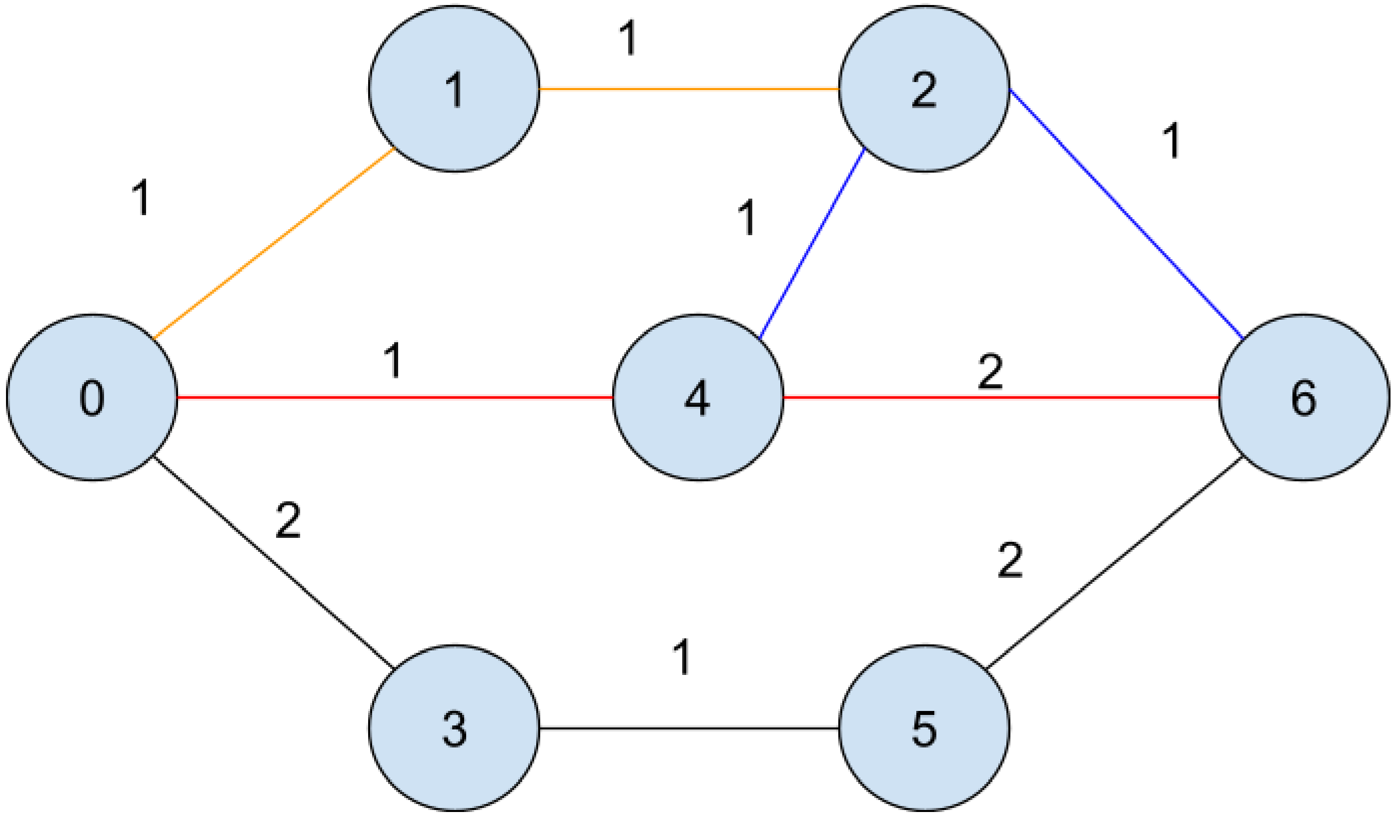
Here the possible paths to reach the end with minimum distance are :
0->6 : 7
0->4->6 : 7
0->1->2->5->6 : 7

0->1->3->5->6 : 7

Therefore we would return 8

We will be using Dijkstra Algorithm with slight modification in it
We can have multiple paths incoming from the same node so we also need to count the number of paths from incoming node

Eg.

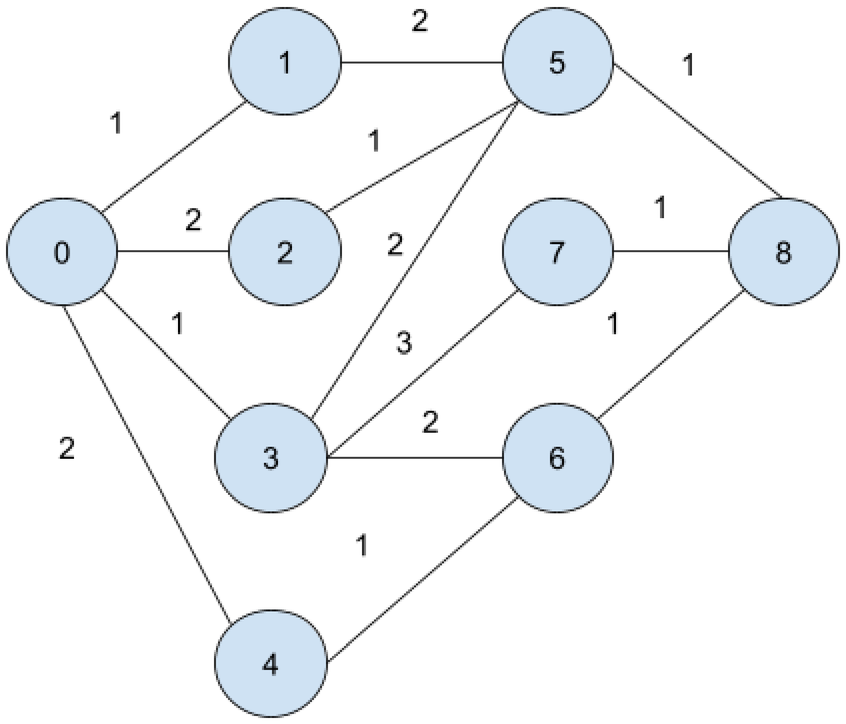


We can see that the 3 can be reached in 2 ways from 2 therefore we can say that the total number of paths can be

numPaths[3] = numPaths[2]+numPaths[4]

We will solve it using min-heap ordered priority queue and with distance array we will also be using ways vector

Eg.



Initial Configuration :

Priority Queue : [distance, node]

[0,0]

Ways :

1	0	0	0	0	0	0	0	0

Distance :

0	inf	inf	inf	inf	inf	inf	inf	inf

Ways :

1	1	1	1	1	3	2	1	5

Distance :

0	1	2	1	2	3	3	4	4

Priority Queue : [distance, node]

[0,0]

[2,4]
[2,2]
[1,3]
[1,1]

[3,5]
[2,4]
[2,2]
[1,3]

[4,7]
[3,5]
[2,4]
[2,2]

[3,6]
[4,7]
[3,5]
[2,4]

[4,8]
[3,6]
[4,7]
[3,5]

[4,8]
[3,6]
[4,7]

[4,8]
[3,6]

[4,8] -> [ ] : We will return ways[n-1] ie. 5

Approach

- Creating an adjacency list
- Creating a min-heap ordered priority queue having pairs of [distance, node]
- Insert the node 0 with distance 0 into the queue
- Create a distance vector and initialize all with infinity
- Mark the 0 node to have a distance 0
- Create a ways vector having all elements initialized with 0
- Mark the 0 node to have no of ways as 1
- Traverse until the queue becomes empty :
    - Extract the first element from the queue
    - Pop the first element from the queue
    - Traverse for all adjacent elements :
        - Check if we reach with a better distance ie. we reach the node for the first time :
            - Update the distance
            - Mark the number of ways to be ways source node is reachable
            - Insert the element with updated distance into the queue
        - Check if the node is reachable with the same distance :
            - Update the number of ways to be a sum of number of ways to reach source + number of ways to reach the node
- Return the number of ways to reach the n-1 node % LONG_MAX

Function Code

```
int countPaths(int n, vector<vector<int>>& roads) {
        // Create the adjacency list
        vector<pair<int, int>> adj[n];
        for(auto it:roads)
        {
            // adding the elements bi-directionally
            adj[it[0]].push_back({it[1],it[2]});
            adj[it[1]].push_back({it[0],it[2]});
        }
        // Creating a priority queue
        priority_queue<pair<int, int>, vector<pair<int, int>>,
greater<pair<int, int>>> pq;
        // inserting the first element with 0 distance into the priority
queue
        pq.push({0,0});
        // creating a distance vector
```

```cpp
        vector<long> distance(n,LONG_MAX);
        // marking the source to have a distance 0
        distance[0] = 0;
        // creating a ways vector
        vector<long> ways(n,0);
        // marking the source to have at least 1 way
        ways[0] = 1;
        // creating a mod variable
        int mod = (long)(1e18+7);
        // traversing until the queue becomes empty
        while(!pq.empty())
        {
            // extracting the first element
            auto it = pq.top();
            long dist = it.first;
            long node = it.second;
            // deleting the first element from the queue
            pq.pop();
            // traversing for the adjacent elements
            for(auto it: adj[node])
            {
                long adjnode = it.first;
                long adjdist = it.second;

                // if we have reached the node for first time
                if(distance[adjnode]>(adjdist+dist))
                {
                    // updating weight
                    distance[adjnode] = (adjdist+dist);
                    // updating number of ways
                    ways[adjnode] = ways[node];
                    // inserting element into queue
                    pq.push({dist+adjdist,adjnode});
                }
                // checking if node is reachable with same distance
                else if(distance[adjnode]==(adjdist+dist))
                {
                    // incrementing number of ways
                    ways[adjnode] = ways[adjnode]+ways[node];
                }
            }
        }
        return ways[n-1]%mod;
```

```
    }
```

Time Complexity

E* log(V)