# Packet Routing

# Part I. Least Cost Path Routing

# Problem Description

From the name itself, it is known that this algorithm finds the path which has the lowest cost in total to reach the destination. The path cost will be considered while finding the paths between the nodes. The main aim of this algorithm is to find the correct path with the lowest path cost.

# Algorithm

1. Initially, create a matrix to store the distance values while traversing the om source to all other nodes.
2. While moving forward, each node should store neighborghbour nodes.
3. Next, the source node should be initialized with 0 value and other nodes sanuld be in infinite value.
4. Create while loop and start when there are unvisited lists.
5. Find the smallest cost between the neighbour nodes.
6. For current nodes of neighbour node.
7. Add edge weight, if the distance of neighbour distance is greater than current node. Then, update distance of neighbour node to the distance of current node add edge weight.
8. Remove the current node from unvisited list and add it to visited node. Return the nodes visited.

# Implementation

## Source Code

//import package

package PacketRouting;

import java.io.\*;

import java.util.\*;

//The class Packet routing

public class Lab4U01037800I {

// allocates an amount of memory needed to store the object

private static int dist[] = new int[100001];

private static int p[] = new int[100001];

// create array object

private static ArrayList<ArrayList<Edge>> edges = new ArrayList<>();

private static final long obj = 1000000000;

private static class Edge {

int to, cost;// Edge Cost

/\*\*

\*

\* Edge

\*

\* @param t the t

\* @param c the c

\* @return public

\*/

public Edge(int t, int c) {

to = t;

cost = c;

}

}

/\*

\*

\* @throws IOException

\*/

public static void main(String[] args) throws IOException {

// read input

BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

// read the first line of input,which contains the no of nodes and links

String[] s = br.readLine().split(" ");

int n = Integer.parseInt(s[0]);

int m = Integer.parseInt(s[1]);

for (int i = 0; i <= n; i++)

// get all edges who are connected to this node

edges.add(new ArrayList<>());

// Initialize the variable

int i = 0;

while (i < m) {

// read the remaining input lines which contain the links

s = br.readLine().split(" ");

int a = Integer.parseInt(s[0]);

int b = Integer.parseInt(s[1]);

int c = Integer.parseInt(s[2]);

edges.get(a).add(new Edge(b, c));

i++;

}

for (i = 0; i <= n; i++)

dist[i] = (int) obj;

// create a priority queue to store the nodes

PriorityQueue<Edge> queue = new PriorityQueue<>(n, new Comparator<Edge>() {

/\*\*

\*

\* Compare

\*

\* @param a the a

\* @param b the b

\* @return int

\*/

public int compare(Edge a, Edge b) {

return a.cost - b.cost;

}

});

queue.add(new Edge(1, 0));

dist[1] = 0;

// check priority queue empty or not

while (!queue.isEmpty()) {

Edge current = queue.poll();

if (current.cost > dist[current.to])

continue;

// find Least Cost Path Routing

for (Edge edge : edges.get(current.to)) {

if (dist[edge.to] > dist[current.to] + edge.cost) {

dist[edge.to] = dist[current.to] + edge.cost;

p[edge.to] = current.to;

queue.add(new Edge(edge.to, dist[edge.to]));

}

}

}

// assign node to cost

int c = n;

// create array list for Least Cost Path Routing

ArrayList<Integer> LCPR = new ArrayList<>();

while (c != 0) {

LCPR.add(c);

c = p[c];

}

// print the number of nodes in the route

System.out.println(LCPR.size());

Collections.reverse(LCPR);

for (int i1 : LCPR)

// print the nodes in the route in increasing order

System.out.print(i1 + " ");

}

}

## Output



# Time Complexity

The following are the time complexity for this algorithm.

Reading input – O(M)

Updating node costs – O(logN)

Traverse between source to destination - O(N)

Overall - O(MlogN+N)

# Test Cases

|  |  |  |  |
| --- | --- | --- | --- |
| Test Cases | Sample Input | Result (Actual) | Result (Expected) |
| Test case 1 | 5 6  1 2 3  1 3 4  2 3 1  2 4 5  3 4 1  4 5 8 | 4  1 3 4 5 | 4  1 3 4 5 |
| Test case 2 | 5 7  1 3 1  1 2 2  2 3 1  3 4 3  1 4 2  4 5 2  3 5 7 | 3  1 4 5 | 3  1 4 5 |
| Test case 3 | 3 2  1 2 3  2 3 1 | 3  1 2 3 | 3  1 2 3 |