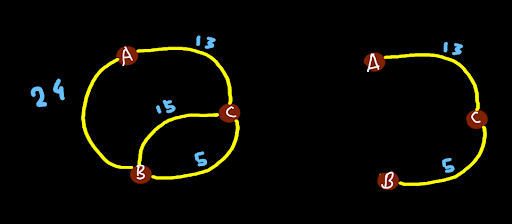
**1. PROBLEM DISCUSSION**

In this problem we will be given a graph which represents a computer network and edges represent the length of LAN wire required to connect two pcs. We have to find the min length of LAN wire required to connect the pcs. We have to print the output in terms of which all pcs need to be connected and the length of wire required to connect the pcs.

**2. APPROACH**

So we want to minimise the cost , it can be achieved by removing the costly edges. But after removing the edge also we want the graph to be fully traversable. In a graph there can be many redundant edges.

****

Like in this graph there are two edges between node B and node C , so for minimising the cost we will remove the edge with weight 15. Similarly we will remove the edge with weight 24 between node A and B. So we want minimum number of edges with which we can connect all the vertices. So what we have achieved in the above graph is also known as minimum spanning tree. So a graph in which all the vertices are connected with a minimum number of edges is known as spanning tree. There can be many spanning trees for a graph so a spanning tree with minimum total edge weight is known as minimum spanning tree. So we will be using Prim's algorithm for finding the spanning tree. So the algorithm is very much similar to BFS but we are going to use Priority Queue instead of queue. Starting from the source vertex at each moment we will pick the edge with minimum cost and add it in out final cost. Rest the algorithm is exactly same as BFS ie. we add the non visited neighbours back in the priority queue. We continue the process till all the vertices are visited.

ConsoleJava

import java.io.\*;

import java.util.\*;

public class Main {

static class Edge {

int src;

int nbr;

int wt;

Edge(int src, int nbr, int wt) {

this.src = src;

this.nbr = nbr;

this.wt = wt;

}

}

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

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

int vtces = Integer.parseInt(br.readLine());

ArrayList<Edge>[] graph = new ArrayList[vtces];

for (int i = 0; i < vtces; i++) {

graph[i] = new ArrayList<>();

}

int edges = Integer.parseInt(br.readLine());

for (int i = 0; i < edges; i++) {

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

int v1 = Integer.parseInt(parts[0]);

int v2 = Integer.parseInt(parts[1]);

int wt = Integer.parseInt(parts[2]);

graph[v1].add(new Edge(v1, v2, wt));

graph[v2].add(new Edge(v2, v1, wt));

}

int src = 0;

PriorityQueue<Pair> queue = new PriorityQueue<>();

queue.add(new Pair(src, -1, 0));

Integer[] visited = new Integer[vtces];

while(queue.size() > 0){

Pair rem = queue.remove();

if(visited[rem.v] != null){

continue;

}

visited[rem.v] = rem.p;

if(rem.p != -1){

System.out.println("[" + rem.v + "-" + rem.p + "@" + rem.wt + "]");

}

for (Edge e : graph[rem.v]) {

if (visited[e.nbr] == null) {

queue.add(new Pair(e.nbr, rem.v, e.wt));

}

}

}

}

static class Pair implements Comparable<Pair> {

Integer v;

Integer p;

int wt;

Pair(Integer v, Integer p, int wt){

this.v = v;

this.p = p;

this.wt = wt;

}

public int compareTo(Pair o){

return this.wt - o.wt;

}

}

}

**3. ANALYSIS**

Time complexity O( (v+e)logv ). Space complexity s=O(v+e)