CS-430

INTRODUCTION TO ALGORITHMS

HOMEWORK #5

Name : MADHUSHALINI MURALI

A ID : A20513784

1)a) Adjacent Matrix

	0	1	2	3	4	5	6
0	0	6	5	5	0	0	0
1	0	0	0	0	-1	0	0
2	0	-2	0	0	1	0	0
3	0	0	-2	0	0	-1	0
4	0	0	0	0	0	0	3
5	0	0	0	0	0	0	3
6	0	0	0	0	0	0	0

1b)

Round-K	$Dist^K[0]$	$Dist^K[1]$	$Dist^K[2]$	$Dist^K[3]$	$Dist^K[4]$	$Dist^K[5]$	$Dist^K[6]$
1	0	6	5	5	∞	∞	∞
2	0	3	3	5	5	4	∞
3	0	1	3	5	0	4	7
4	0	1	3	5	0	4	3
5	0	1	3	5	0	4	3
6	0	1	3	5	0	4	3

Order of edges for Bellman-Ford Algorithm:

(5,6), (4,6), (3,5) (3,2) (2,4) (2,1) (1,4) (0,3) (0,2) (0,1)

1c)

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class Algorithm {
    static int value = Integer.MAX_VALUE;

    public static void BellmanFordAlgorithm(int[] vertex, List<int[]> edges, int
zero_val) {
        int[] dist = new int[vertex.length];
```

```java
        Arrays.fill(dist, value);
        dist[zero_val] = 0;

        for (int i = 1; i < vertex.length; i++) {
            for (int[] edge : edges) {
                int a = edge[0];
                int b = edge[1];
                int weight = edge[2]; // set weight to 1 for unweighted graph
                if (dist[a] != value && dist[a] + weight < dist[b]) {
                    dist[b] = dist[a] + weight;
                }
            }
        }
        System.out.println("Shortest paths from vertex 0:");
        for (int i = 1; i < vertex.length; ++i) {
            System.out.println("Vertex " + i + ": " + (dist[i] == value ? "value"
: dist[i]));
        }
    }

    public static void main(String[] args) {
        int[] vertex = new int[] { 0, 1, 2, 3, 4, 5, 6 };
        List<int[]> edges = new ArrayList<>();
        edges.add(new int[] { 5, 6, 3 });
        edges.add(new int[] { 4, 6, 3 });
        edges.add(new int[] { 3, 5, -1 });
        edges.add(new int[] { 3, 2, -2 });
        edges.add(new int[] { 2, 4, 1 });
        edges.add(new int[] { 2, 1, -2 });
        edges.add(new int[] { 1, 4, -1 });
        edges.add(new int[] { 0, 3, 5 });
        edges.add(new int[] { 0, 2, 5 });
        edges.add(new int[] { 0, 1, 6 });

        BellmanFordAlgorithm(vertex, edges, 0);
    }
}
```

Output:

```java
public class Algorithm {
    static int value = Integer.MAX_VALUE;

```

PROBLEMS 1 OUTPUT DEBUG CONSOLE TERMINAL

```
PS C:\Users\mural> cd "c:\Users\mural\OneDrive\Desktop\" ; if ($?) { javac Algorithm.java } ; if ($?) { java Algorithm }
Shortest paths from vertex 0:
Vertex 1: 1
Vertex 2: 3
Vertex 3: 5
Vertex 4: 0
Vertex 5: 4
Vertex 6: 3
PS C:\Users\mural\OneDrive\Desktop> 
```

2) We can assume that if all the edges weights of an undirected graph are all positive, then any subset of edges that connects all vertices and has maximum total weight is not a tree.

This implies that the subset of edges contains a cycle.

Lets say that, the shortest path,

$p = (v_0, v_1, \ldots\ldots, v_n)$ contains positive cycle.

Consider cycle $cy = (V_a, V_{a+1}, \ldots V_b)$ and $a = b$.

Since it is a positive cycle, we can say that

$$\boxed{w(cy) > 0}$$

If we remove the cycle 'cy' from the path, then we will get

$p' = (v_0, v_1, \ldots, v_a, v_{b+1}, v_{b+2}, \ldots, v_n)$.

We know that $w(p') = w(p) - w(cy)$

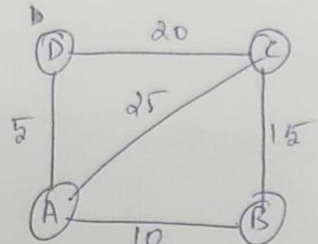
we get $\boxed{w(p') < w(p)}$

But this contradicts our initial assumption that it is a cycle.

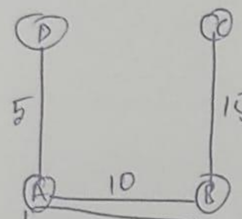
Since, this is a contradiction, we can say that this subset is a tree.

Therefore, if all the edge weights of an undirected graph are positive, then any subset of edges that connects all vertices and has minimum weight is a tree.

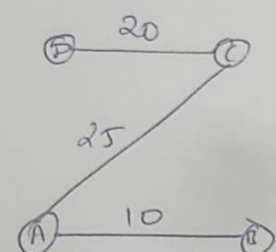
Let us consider the following undirected graph.



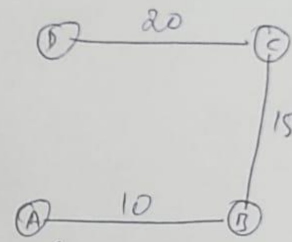
The subgraphs are



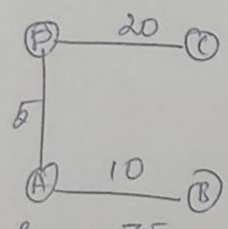
$\boxed{\text{Sum} = 30}$



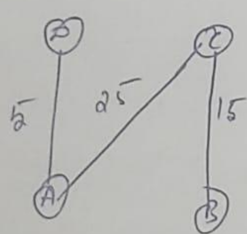
sum = 55



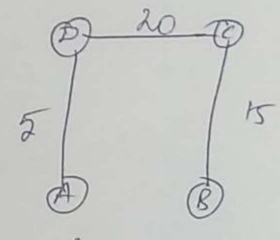
sum = 45



sum : 35



sum = 45



sum = 40

The first subgraph has the minimum value.
Hence the subset of edges that connects all vertices and has minimum total weight.

3) Let us use Breadth First Search algorithm for this problem.

<u>Algorithm</u> : $\left\{ \begin{array}{l} T = \text{True} \\ F = \text{False} \end{array} \right\}$

```
function BFS (G, u, v):
    for each vertex u' ∈ V[G] - {u} :
        do visited [u'] = F

    Qu = new Queue ()
    Qu. enqueue (u)
    visited [u] = T

    while Qu ≠ ϕ      # Running until queue is not empty
        current = Qu.dequeue ()
        if current == v :
            return true
        for each v' ∈ Adj [u'] :
            if not visited [v'] :
                Qu. enqueue (v')
                visited [v'] = T

    return false
```

Time Complexity : $O(|V| + |E|)$

V = number of vertices

E = number of edges

It is because, the program will run for V times and will loop through all Edges in second loop.

Hence time complexity is $O(|V| + |E|)$.

4)
```
function timeConflict(enrolled_events):
        set_conflict = false
        for each a in enrolled_events :   (for 1 event)
            for each b in enrolled_events :  (for all other events)
                if a == b :
                        continue
                if connected (a,b) == false :  (conflict check)
                        set_conflict = true
                        break
            if set_conflict :
                    break
        return not set conflict        (return true
                                         if there's no conflict)

    function connected (e1, e2) :
        if (e1, e2) or (e2, e1) in E :  (# E = edge)
                return true (return true if conflict
                             is there)
        else :
                return false
```

The connected graph is used to check whether if there is any edge is present~~ed~~ inbetween 2 events, if so, then there is no conflict.

The algorithm scans each pair of event in the enrolled_events and returns true or false value accordingly.