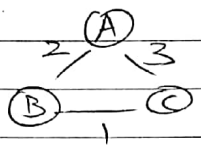
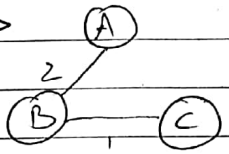
Applied Algorithms HW 5

① To find the maximum spanning tree we do the inverse of minimum spanning tree. Since Kruskal's Algorithm, works for negative edge weights too, we negate the weights of the graph and run Kruskal's Algorithm to find the maximum spanning tree.

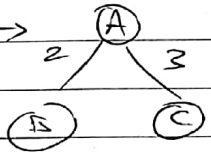
For eg:-



Min spanning tree →



Max spanning tree →



So for input we can take a adjacency Matrix of cost

	A	B	C
A	0	2	3
B	2	0	1
C	3	1	0

Algorithm

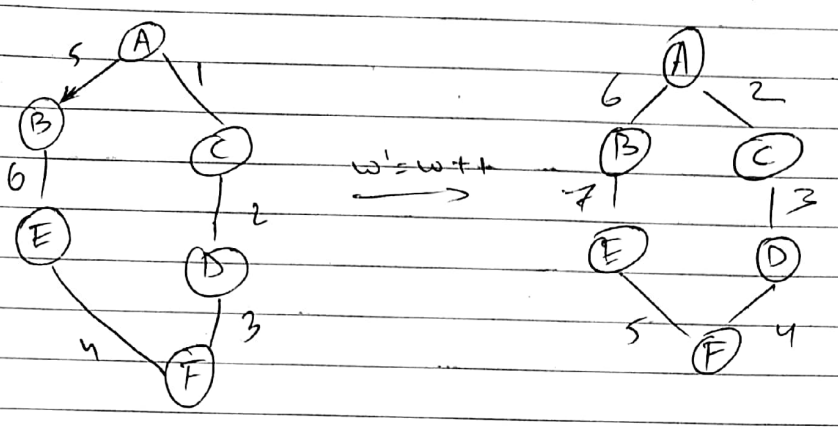
```
Main ()
{
    cost [row] [col] = -(< adjacency Matrix input >) // negate all the edge weights
    row = col = <no. of nodes in the graph>
    int ne = 1;
    int parent [row];

    while (ne < row)
    {
        int Min = < Integer. smallest negative number >
        for (i = 1 to i = n)
        {
            for (j = 1 to j = n)
            {
                if (cost [i] [j] < Min)
                {
                    Min = cost [i] [j]
                    a = u = i
                    b = v = j
                }
            }
        }
```

```
u = find(u);
v = find(v);
if (search(u,v))
{
        print("edge", a, "->", b, "=", Min);
        Mincost += Min;
        ne++;
}
cost[a][b] = cost[b][a] = <smallest negative integer>
print("total mincost", Mincost);
}
int find(int i)
{
        while(parent[i])
        {
            i = parent[i];
        }
        return i;
}
int search(int i, int j)
{
        if(i != j)
        {
            parent[j] = i;
            return 1;
        }
        return 0;
}
```

Brv Time complexity = $O(E \log E + V \log V)$
$= O(E \log E)$ OR $O(E \log V)$

Kruskal's algorithm Referenced from:- "scontreetree.com/data_structure/kruskal-algorithm".

(2) lets consider this example for the given condition



(a) Minimum spanning tree does not change.

in case (I) : Min Span tree :- A—B
A - C
C - D
D - F
F - E

in case II : Min Span tree :- A - B
A - C
C - D
D - F
F - E

~~State all the~~

Since all the weights are increased equally adding a constant to all the weights in Kruskals does not effect the final nodes only the ∄ aggregate weights are effected.

(b) The shortest path changes

case (I) : Shortest path from A → E ⇒ A - C - D - F - E
⇒ 1 + 2 + 3 + 4
⇒ 10.

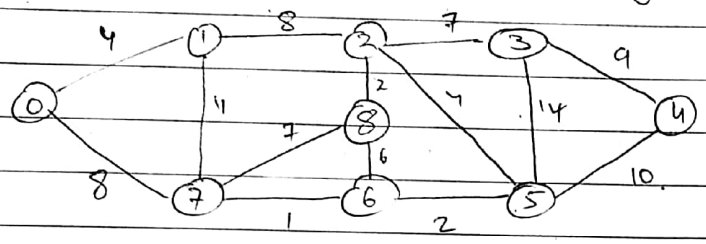
case (II): shortest path from A → E ⇒ A - B - E
⇒ 6 + 7
⇒ 13.

Q-3]

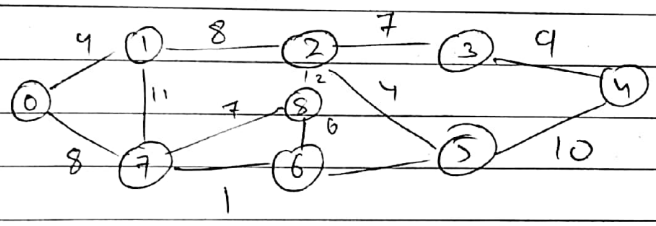
The given algorithm is a greedy algorithm that deletes the most weighted edge of the graph at each iteration if the graph is still connected.

The algorithm is a MST algorithm we can prove it with the given example

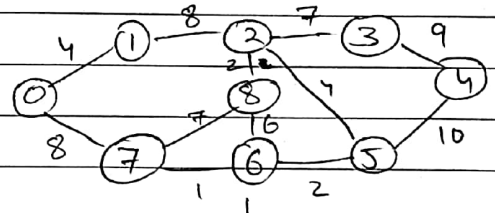
lets consider the graph from (https://www.geeksforgeeks.org/kruskals-minimum-spanning-tree-algorithm-greedy-algo-2/)



Highest edge = 14 (Remove)



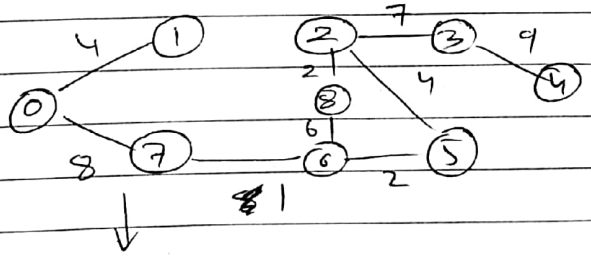
Highest edge = 11 (Remove)

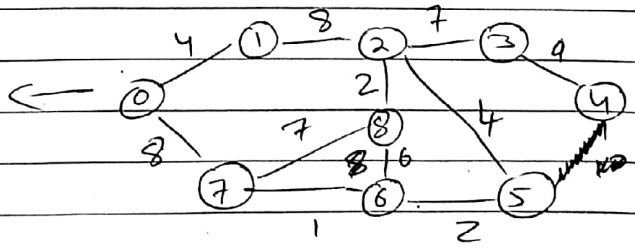


Highest edge = 10 (Remove)

Highest edge = 9 (cannot)

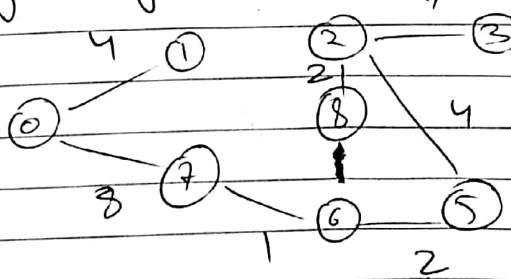
Highest edge = 8 (Remove)





Highest edge = 7 (cannot)

Highest edge = 6 (Remove)

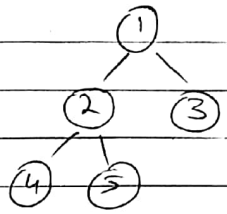


(Final MST)

This is the same output we will get from Kruskal hence it is a verified MST.

(4) The diameter of the ~~graph~~ tree can be found out using DFS

Algorithm




```
Main ()
{
    list adj = [ ]

    adj[1].add(2)
    adj[2].add(4)
      :
      :

// Make the tree using adj List.

diameter (adj, n)
}
diameter (adj, n)
{
    Max_count = Integer.MIN_VALUE
    dfs (1, n, adj) ;    // find farthest node from root
    dfs (x, n, adj)      // find farthest node from x
    return maxcount
}
dfs (int node, int n, list adj[ ])
{
    boolean [ ] visited = [ ] , count = 0
    visited [1 ... n] = false
    dfsutil (node, count+1, visited, adj)
}
dfsutil (int node, int count, boolean visited[ ], list adj[ ])
{
    visited [node] = true
    count++;
    list l = adj [node]
    for ( elements in l)
    {
        if (!visited[i] ~~count = maxcount~~ ){ if(count >= Max count)
        { Max count = count;
          x = i;
```

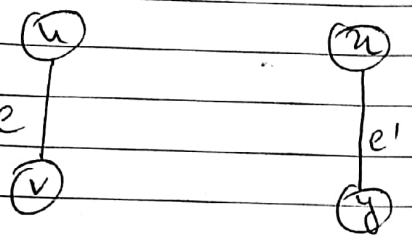
```
        }
    dfsUtil(i, count, visited, adj)
    }
  }
 }
}
```

Time complexity = $O(\log(|V| + |E|))$ → DFS time complexity.

References:- "https://www.geeksforgeeks.org/diameter-tree-using-dfs"

Q-5]

lets consider the following graph



~~So out~~

Algorithm

→ start from node (u)

→ go to node (v) total cost = e

Now from (v) we find the minimum distance to x and minimum distance to y using dijkstras

∴ we get v-x, v-y — total distance

select ~~it~~ cost = Min(v-x, v-y)

∴ total cost = e + Min(v-x, v-y)

→ Now take the edge e'

∴ total cost = e + Min(v-x, v-y) + e'

→ Now if v-x is selected find dis(x-u) ~~and~~

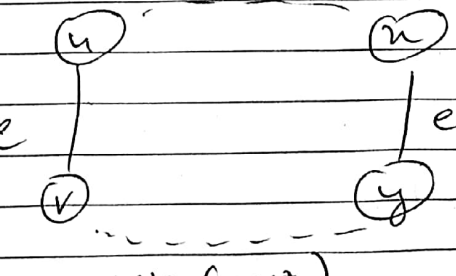
if v-y is selected find dis(y-u)

using dijkstras we can find the minimum distance.

∴ total cost = e + Min(v-x, v-y) + {dis(y-u) or dis(x-u)}

conditioned on what is selected in Min(v-x, v-y)

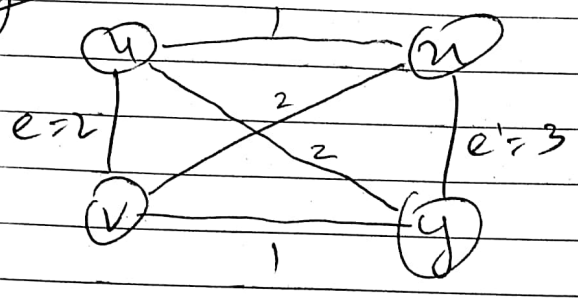
Min(cost)



Min (cost)

If at any point of time dijkstras returns no path the no cycle exists

Eg:-



$u \rightarrow v$ cost = 2

$\min(v \rightarrow x, v \rightarrow y)$

$= v \rightarrow y$ cost = 2+1

$y \rightarrow x$ cost = 2+1+3

$x \rightarrow u$ cost = 2+1+3+1

∴ Total min weight = 7.

edges :- $u - v - y - x - u$.

Total Run time = $O(V^2) + O(V^2) + O(V^2)$

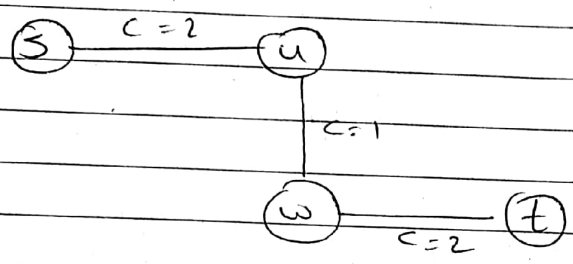
$= O(V^2)$

Since we are Running the shortest path dijkstra's algorithm thrice.

Q-6]

a) False

The counterexample for this is:-



The capacity of s-u = 2

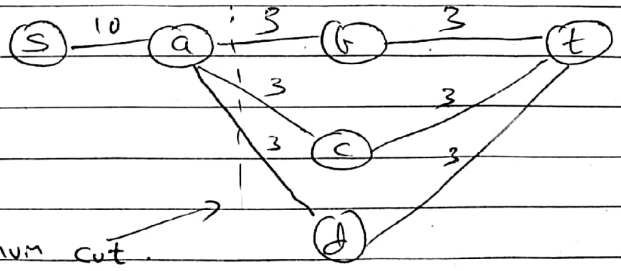
u-w = 1

w-t = 2

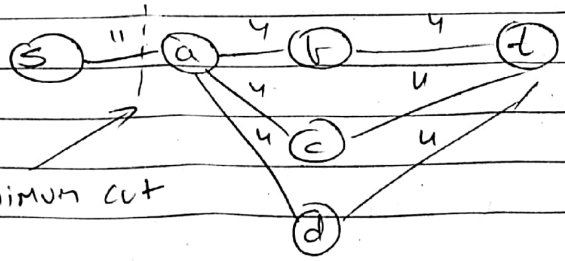
So out here the maximum flow has a value of f = 1 In this case u-w edge does not get saturated out of the sink 't'

b) False.

lets consider the example



Now increase the edge weights by 1



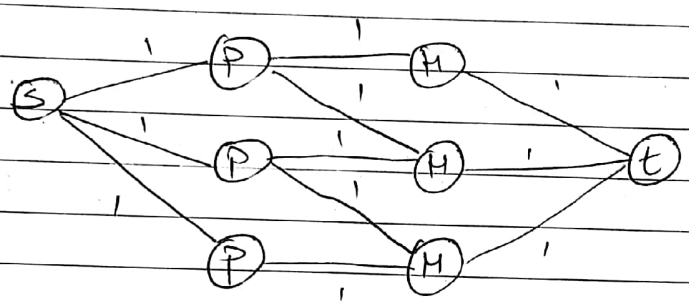
The edge changes when the weights are increased by 1.

Q-7)

The given problem can be formulated as a Maximum - flow problem

We construct a graph with the nodes p as patients, node h as hospital and a source and sink

lets consider 3 patients and 3 hospitals, connect the patients to the nodes and hospitals to the sink



The cost from $S \rightarrow p = 1$

$\therefore$ The cost from $h \rightarrow t = n/k = 3/3 = 1$

Now lets find the maximum flow from the above network flow diagram using ford-fulkerson algorithm

Referenced from :- https://en.wikipedia.org/wiki/Ford-Fulkerson_algorithm

1. $f(u,v) \leftarrow 0$ for all edges (u,v)
2. while there is a path p from s to t in Graph G such that $c(u,v) > 0$ for all edge (u,v) in p.
    1. find $c_f(p) = \min\{c_f(u,v) : (u,v) \in p\}$
    2. for each edge $(u,v) \in p$
        1. $f(u,v) \leftarrow f(u,v) + c_f(p)$
        2. $f(u,v) \leftarrow f(u,v) - c_f(p)$

return $f(u,v)$

if (maxflow == n) : the problem can be solved and its possible to take the people to the hospital without any overflow.

if (maxflow < n) : then the problem cannot be solved

Runtime $\Rightarrow O(V+2E)$

where $V \rightarrow n+k+2$
$E \rightarrow nk+n+k$.

where n = patients
k = hospitals
2 → source & sink.

Q-8]

Approach 1

→ we can solve this problem wing greedy Algorithm

→ First we assign all the variables the same value and then according to the equality constraints we test whether any of the inequality constraints are violated

→ Next we assign all the variable its own unique set. Then we join all the sets that are equal. This can be done by a Union-Find structure from Kruskal's Algorithm.

→ if $(x_i = x_j)$ and if $(Find(x_i) = Find(x_j))$ → its correct meaning equal constraints are rightly grouped together in the same set.

→ If they are not in the same set then do a union on x_i, x_j Union (x_i, x_j)

→ The union operation → $O(n \log n)$

→ After all the sets are grouped together accordingly we scan through the list of disequality constraints The constraints can only be satisfied if $x_i \neq x_j$ there is $Find(x_i) \neq Find(x_j)$

→ This linear scan time takes → $O(m)$ time

→ Hence total Run-time of Algorithm = $O(m + n \log n)$

Approach 2

→ We start by building a graph where each ~~edge is the~~ node is the variable

→ We connect all nodes where $x_i = x_j$

→ Now we compute the connected components of graph in linear time and check if $x_i \neq x_j$ to determine if x_i and x_j are in the same connected component.

→ So for this we can use Union (x_i, x_j) to add edges and Find (x_i, x_j) to compare the components. These have a constant time, and we do not keep checking again and again for the sets ie. the condition if $(x_i = x_j)$ & Find $(x_i = x_j)$ "true". This makes the algorithm linear

→ Total time complexity = $O(n+m)$

References:- https://web.cs.dal.ca/~whidden/CS3110/assignments/alt_solutions.pdf