Assignment 5: Srinithish

Question 1

Show how to find the maximum spanning tree of a graph, which is the spanning tree of the largest total weight.?

The solution for finding the maximum spanning tree as same as the minimum spanning tree except that the edges are to be sorted in decreasing order when chosen to form the Maximum spanning tree.

```
MaxST(G,eW): ## ew is the edge weights
    A = {} ## empty set

    for each vertex v in G.V:
    ## make a disjoint set where the parents are directly retrieved
        Make-set(v)

    sort the edges of G.E into decreasing order by weight w

    for each edge (u,v) in G.E:  ##considered decreasing order

        if FIND-SET(u) != FIND-SET(v):

                A = A + {(u,v)} ##include in the maximum set
                Union(u,v) ## make parents of u,v point to the same

    return A
```

Question 2

Consider an undirected graph G = (V, E) with nonnegative edge weights we. Suppose that you have computed a minimum spanning tree of G, and that you have also computed shortest paths to all nodes from a particular node s. Now suppose each edge weight is increased by 1; the new weights are w'e = we + 1.

1. Does the minimum spanning tree change? Give an example where it changes or prove it cannot change.

The minimum spanning tree doesnot change for the following reasons, consider the KRUSKAL algorithm, the order of the edges condiered and FIND-SET are the most important parts for the algorithm to execute and form a MST

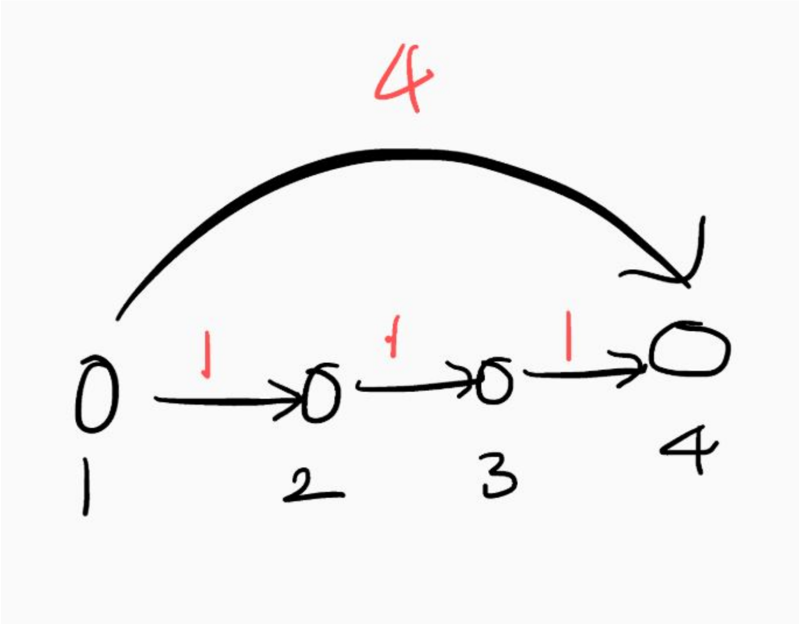
1. When a constant c is added to every edge , the sorting order is not disturbed and hence the order remains the same as the original Graph

2. And the step FIND-SET is also unchanged as the parents of original Vertex are still the same in the new modified graph.

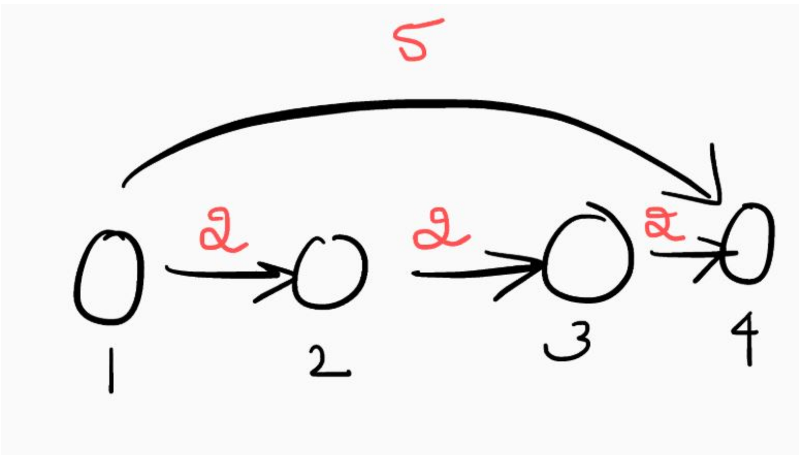
Hence the MST remains the same but the Minimum Value is offset by the c*V

2. **Do the shortest paths change? Give an example where they change or prove they cannot change?**

Yes the shortest paths change as in below the example



When 1 is added to each node the shortest path changes from 1 -> 2 > 3 -> 4 to 1 -> 4



Question 3

The algorithm will give a Minimum Spanning Tree becuase at each iteration we are only removing a edge if the residual graph is still connected , in case if we had the edge in the graph still it would be connected but with a more weight, we might as well remove the edge which is giving additinal edge weight to the graph. Also we are only removing the edge if the removal doenst render the trees disconnected In spanning tree we just need to ensure that it is connected and has minimum weights, We are keeping this invariant intact at every iteration as weight of the tree having this edge included in the graph is higher than if removed

Question 4

Description

1. Pick a random vertex 's' and find do a BFS search starting from this vertex
2. With this you'll find shortest segmental distance (simple paths) to all the nodes.
3. Choose vertex whose distance from the start vertex was the maximum say 'b'
4. Now 'b' as start vertex do a BFS and find the vertex whose distance from b is maximum.
5. Now distance of 'a-b' is the diameter of the Tree T

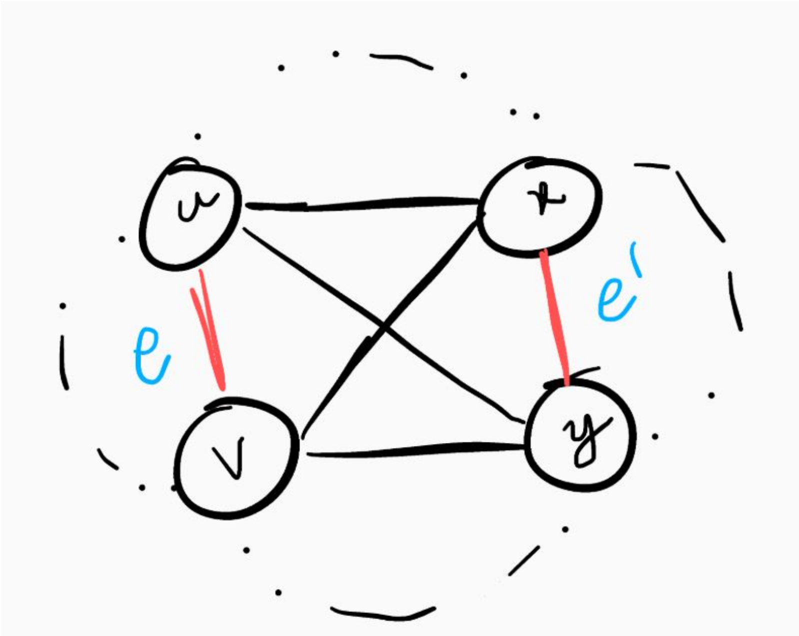
In [0]:

```
import queue
 ##pick a random vertex

def BFS(Graph,start_node):

    Q = queue.Queue()
    Q.put((start_node,0)) ##node and the distance from start_node

    dictOfDistances = {}

    while Q.empty() != True:
        (p,parent_dist) = Q.get()
        p.visited = True

        for neigh in p.neighbours:

            neigh_distance = parent_dist + 1 ##one segment added

            if neigh.visited == False:
                Q.put((neigh,neigh_distance))

            dictOfDistances[neigh] = neigh_distance
    return dictOfDistances


s = random(V)
AllDistFromS = BFS(Graph,s) ##get the distance to all nodes from s

a_node,distance  = max(AllDistFromS, key = distFromS[i]) ## get the node that coresponds tc

AllDistFrom_a = BFS(Graph,a_node) ## set this node as the start node for the BFS

b_node,Finaldistance = max(AllDistFrom_a, key = distFromS[i]) ## get the end node and distc
```

Above **a_node --> b_node** is the max path and the diameter is the '**Finaldistance**'

Since BFS take O(V+E) time and is called twice, Complexity is (V+E)

Question 5



!

Description:

Consider the above graph Let C be a cycle in the graph which contains edge e and e', These can be cycles in following configuration

1. Vertex **u** connects to **x** then **v** connects to **y** , note that this connection need not be direct it jus means that there is a path
2. Vertex **u** is connected to **y** and **v** connects to **x**

Now the algorithm,

1. Remove edges e and e' from the graph and run dijsktra with source as 'u' We ll have min distances d to 'y' and 'x' from 'u'
2. Run the dijkstra with 'v' as the source to 'x' and 'y' , we'll have min distances d' to x and y from v
3. In the first configuration when u --> x and v --> y , the total cycle weight = a = w(e) + w(e') + x.d + y.d'
4. In the first configuration when u -->y and v --> x , the total cycle weight = b = w(e) + w(e') + x.d' + y.d

   Then the weight of the shortest cycle containing e and e' is min(a,b)

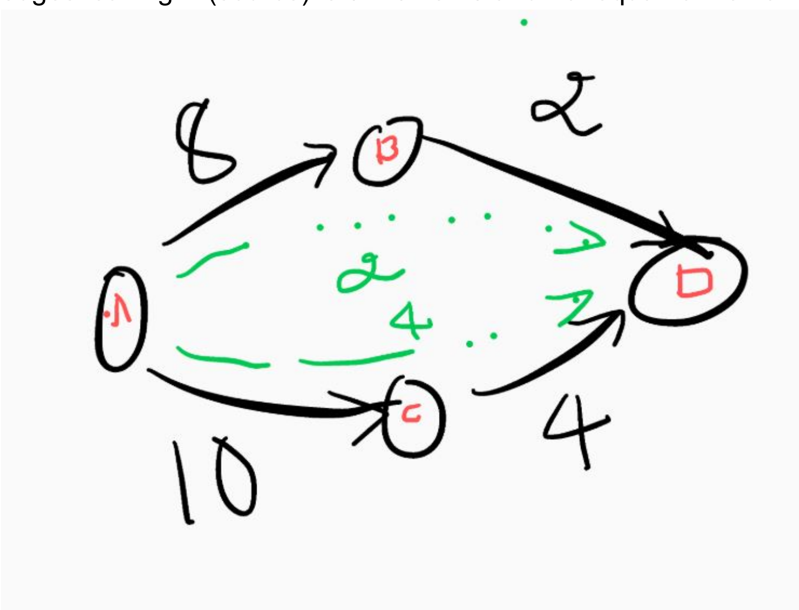
The complexity of algorithm is equal to dijkstra $O(V^2 * E)$

In [0]:

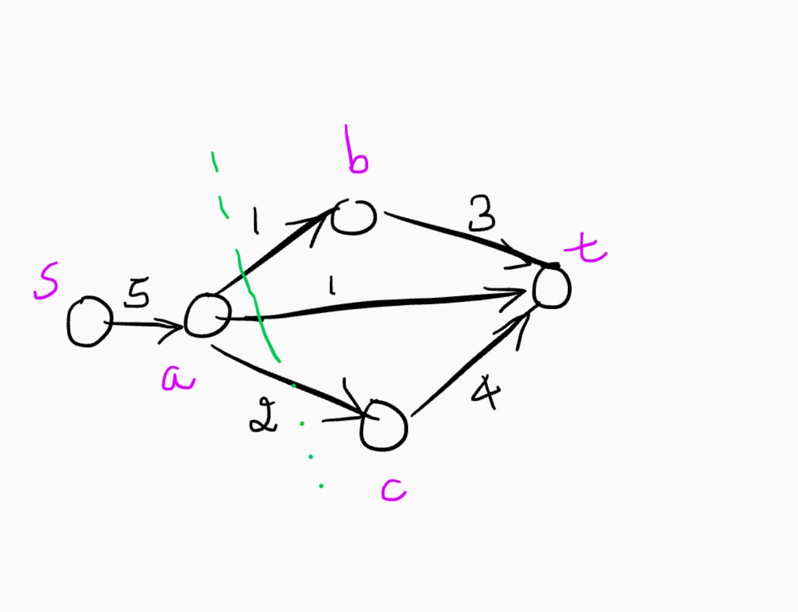
```
def findShortestCycle(Graph,(u,v),(x,y)):

    Graph.remove((u,v)) ## removes the edge e (u,v)
    Graph.remove((x,y)) ## removes the edge e' (x,y)

    dijkstra(Graph,u) ## find min distances from u to all vertices

    u_to_x = x.distance
    u_to_y = y.distance

    ## initialise all d to zero
    dijkstra(Graph,v)  ## find min distances from u to all vertices

    v_to_x = x.distance
    v_to_y = y.distance


    cycle_Weight1 = Graph.weight((u,v)) + Graph.weight((x,y)) + u_to_x + v_to_y
    cycle_Weight2 = Graph.weight((u,v)) + Graph.weight((x,y)) + v_to_x + u_to_y


    minCycleWeight = min(cycle_Weight1,cycle_Weight2)

    if minCycleWeight != float('Inf')

        return minCycleWeight

    else:
        return False
```

Question 6

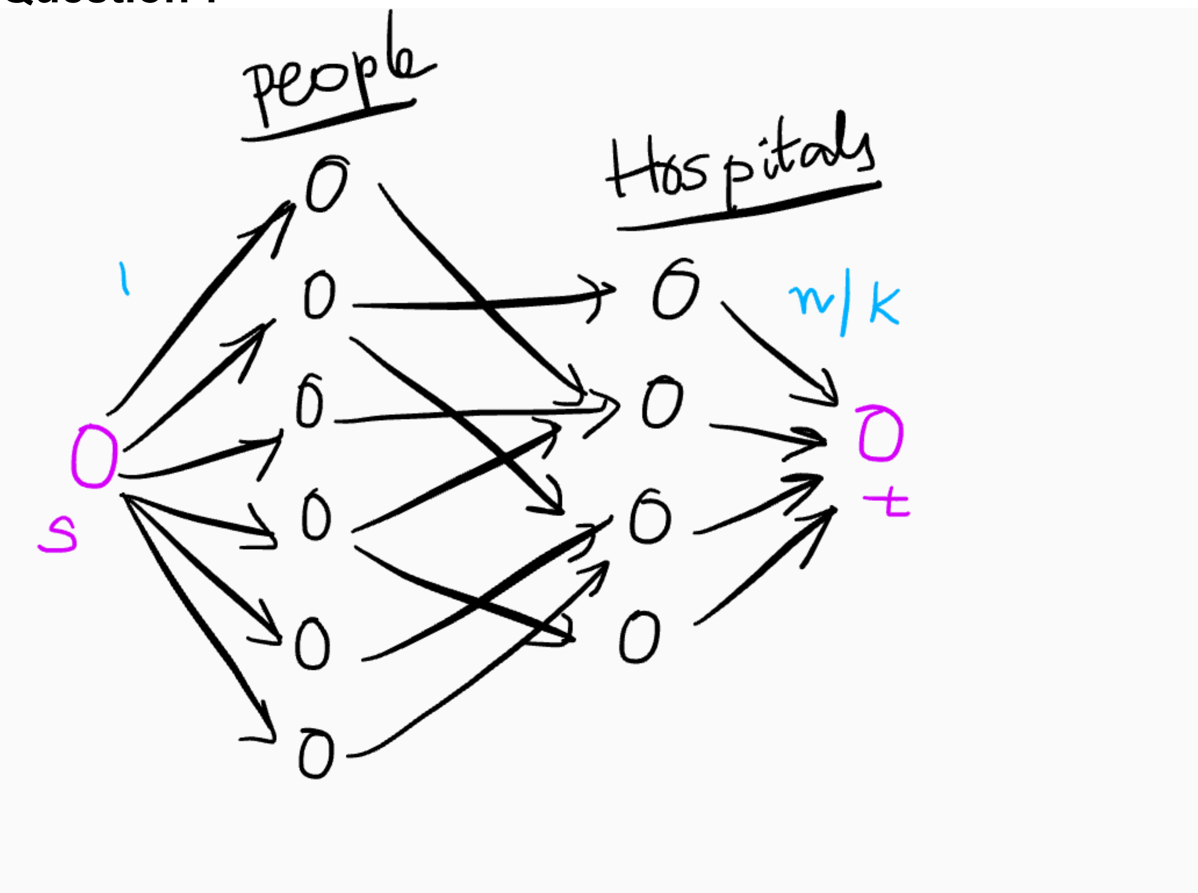
a. In the below graph you can see that the flow is 2 + 4 (denoted in green) i.e 6 however the total capacity of the edges leaving A (source) is 8+10 i.e 18 and not equal to the flow. Hence disproved.



b. The statement is false. Consider the below graph, initially the minimum cut passes through edges (a,b) and (a,c) When all the edges are added with 1 the minimum cut is edge (s,a). Hence disproved



Question 7



1. In the graph let the people be denoted by nodes on the left in fig p_nodes
2. Let hospitals be denoted by nodes on the right in the fig h_nodes
3. Let the edges denote possibility of a person going to a certain hospita i.e edge (p,h) denotes hospital h is in the allowed range of the person p and p can visit it.
4. If there are hospitals h farther than allowed distance for a person p then there exists no edge from p to h
5. This problem reduces to solving bipartite graph.
6. Create a dummy node s as source and t as sink
7. Connect all the people nodes to source as shown in the figure

8. Connect all the nodes from hospital to sink whose capacity each is **n/k** and assign the same.
9. Assign a capacity of **1** to all the rest of the nodes
10. Now solve for the maximum flow using ford_fulkerson , if the max flow 'f' happens to be n then this assignment of people to hospitals is possible else its not

The complexity of the algorithm is O(E*C)

In [0]:

```
def checkFeasibility(Graph,n_people,k_hospitals):
    maxFlow = Ford_Fulkerson(Graph) ##solve for max flow

    if maxFlow == n_people :
        return True
    else :
        return False
```

Question 8

I arrived with the below solution with help of Swarnima Sowani We can solve this BFS assigning appropriate sets based on equaliteies and inequalities. Say we have set1 and set2 where set 1 has all elements with equal values and set 2 has all elements with value not equal to any variable from set 1.

Let each of the variable x_1, x_2 be vertices in the graph and let the constrains be denoted by the edges between them. Let all equlities be denoted as edges with weight 1 and inequalities be denoted with weight -1

say if $x_1 = x_2$ then edgeweight of (x_1, x_2) is 1

say if $x_1 \neq x_2$ then edgeweight of (x_1, x_2) is -1

Starting with one node and applying BFS, we will explore all the neighbors and assign same set for equal variables and opposite sets for unequal variables. If some variable has already been assigned a set which is not following the constraint then return false else return True

Example: For example in question, We will start with node x1 and assign it to set1 and insert it to queue. While queue not empty: Take the element in queue which is x1, and check all its neighbors.

Since x1=x2 and x2.set = None, x2.set = x1.set that is both x1 and x2 are assigned to set1 Then x1 != x4 and x4.set ==null so, x4 is assigned to set opposite to x1 that is set2. There are no more neighbors of x1 so start with next variable in queue that is x2. x2 has neighbor x3 with equality constraint and x3 has not assigned any set. So, set of x3 will be same as x2. x2 has no more neighbors so we will move with next element in queue that is x3. x3 has only one neighbor x3=x4 but x4 is assigned to set2 which is not same as the set of x3. Hence the constraints are violated.

In [0]:

```
def isSatisifying(Graph):

    Q = queue.Queue()
    start_node = random(Graph.Vertices)
    Q.put((start_node,0)) ##node and the distance from start_node

    ##initialise all vertex set to None
    for v in Graph.Vertices:
        v.set = None

    start_node.set = 1

    while Q.empty() != True:
        (p,parent_dist) = Q.get()
        p.visited = True


        for neigh in p.neighbours:

            sign = Graph.weights((p,neigh))

            if neigh.set == None: ##if not already assigned the set
                if sign == -1:  ##if its inequality
                    neigh.set = -1*p.set  ##give the opposite set
                if sign == 1:
                    neigh.set = p.set ##else same set


            if neigh.set != None: ##if already a set is assigned

                if sign == 1: ##if its equality

                    if neigh.set != p.set: ##if its parent and neigh do not agree
                        return False
                if sign == -1:

                    if neigh.set == p.set:
                        return False

            if neigh.visited == False:
                Q.put((neigh,neigh_distance))
```