Homework 5

- Amit Makashir

Q1.

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

Solution:

In Kruskal's algorithm, we sort the edges in ascending order to build the spanning tree. If we reverse this step and instead sort the edges in descending order to build the spanning tree, the resulting spanning tree will be a maximum spanning tree.

The pseudocode for this algorithm is given below:

V => a list of all the vertices of this graph E => a list of tuples containing the vertices and edge weight (u_i, v_i, w_i)

```
def Max_spanning_tree(V,E):
    E = sort_desc(E)  # sort on w for each weight
    A = []

    for u,v,w in E:
        # Add this edge only if their parents are different
        # Otherwise you'll get a cycle
        if Find(u) != Find(v):
            A.append((u,v,w))
            # Combine these trees, make their parents same
            Union(u,v)
    # A is the list of edges to be included in the maximum spanning tree
    return A
```

Q2

Consider an undirected graph G = (V, E) with nonnegative edge weights w_e. 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 = w_e + 1.

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

(b) Do the shortest paths change? Give an example where they change or prove they cannot change.

Solution:

a)

The minimum spanning tree will change if the order of edges change. As we are incrementing the weight of edges by 1, the sorted order will remain the same and hence *the minimum spanning tree will not change*.

b)

It is possible that the shortest paths to change. An example for such a scenario would be as follows:
Let the edge e_1 be $(u, v, 4)$ .... $(node1, node2, edgeweight)$
Let the edge e_2 be $(u, x, 1)$
Let the edge e_3 be $(x, y, 1)$
Let the edge e_4 be $(y, v, 1)$

The shortest path from u to v is is through x and y
(u -> x -> y -> v) => (Total weight: 1+1+1 = **3**)
(u -> v) => (Total weight: **4**)

However, after incrementing by 1, we can see that the shortest path from u to v is through edge e_1.
(u -> x -> y -> v) => (Total weight: 2+2+2 = **6**)
(u -> v) => (Total weight: **5**)

Q3

Either prove that the output of the algorithm is a minimum-spanning tree or give a counterexample when the algorithm does not output a minimum-spanning tree.

Solution

This is similar to a problem discussed in class, where we wanted to know if a particular edge will be present in atleast in one of the MSTs. In this problem, we remove edges in decreasing order. If after removing this edge, the graph is connected, it means that there is a path to all the nodes without this edge. As all the edges in the connected graph are smaller than this edge, we can safely say that after removing all such large non-essential edges we will get a MST. Hence, the output of the algorithm is a MST.

Q4

The diameter of a tree T=(V, E) is defined as the largest length of all shortest path (i.e., the path with the smallest number of edges) between pairs of vertices. Devise a O(|V| + |E|) algorithm to compute a diameter for a given tree.

Solution:

The algorithm can be devised as follows:

1. Pick a node and start a BFS traversal from it (keep visited checks) This will give us the shortest distance to all leaf nodes from the start node.
2. Once all the graph is explored, find the leaf node with maximum distance from our source/start node (Say "v").
3. Start a BFS traversal from this node "v" and record the distances to reach the leaf nodes.
4. The leaf node with maximum distance from node "v" is the largest length of all the shortest path

max(BFS(s)) -> v
max(BFS(v)) -> u
v -> u is the diameter of the tree

Time complexity: As we performing two serial BFS traversal, the time complexity would be $O(|V| + |E|)$

Nithish Kandagadla helped in answering this question

Q5

Let G be an arbitrary connected, undirected graph with a distinct positive weight on each edge. Let e=(u,v) and e'= (x, y) are two arbitrary edges connecting four distinct vertices u, v, x and y. Devise a polynomial time algorithm to find a cycle in G with the lowest total weight that contains both e and e'. What is the running time of your algorithm.

Solution:

This can achieved by the following algorithm:

Say we have G(V,E)

1. Remove the edge e from the graph. (E = E - $\{e\}$)
2. Using dijkstra's shortest path algorithm, calculate the distance from u to x and y.
3. Similarly, calculate distance from v to x and y.
4. Let a = dist(u,x) + dist(v,y)

```
b = dist(u,y) + dist(v,x)
```

5. c = min(a,b) + w_e + $w_{e'}$
6. c is the lowest total weight that contains both e and e'

As we running Dijkstra's two times serially, the time complexity of this algorithm is $O(E.log(V))$

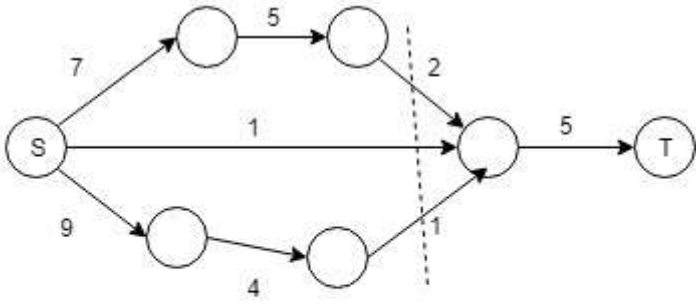
Q6

a)

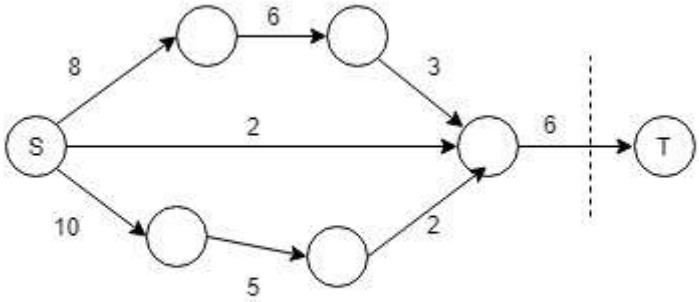
In the fig(a), maximum flow of the network is 2+1+1 = 4, but this doesn't saturate edges with 7,9 and 5. Therefore, the statement is false.

b)

In the fig(a), the minimum cut is 2+1+1 = 4, but after adding 1 to every edge, the minimum cut becomes 6. Therefore, the statement is false.



Fig(a)



Fig(b)

Q7

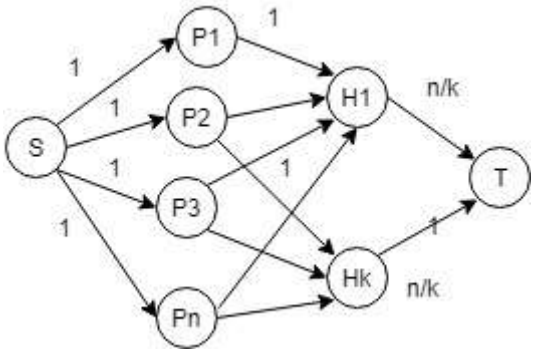
Consider the following scenario. Due to large-scale flooding in a region, parametics have identified a set of n injured people distributed across the region who need to be rushed to hospitals. There are k hospitals in the region and each of the n people needs to be brought to a hospital that is within a half-hour's driving time of their current location (so different people will have different options for hospitals, depending on where they are right now). At the same time, one does not want to overload any one of the hospitals by sending too many patients its way. The parametics are in touch by cell phone, and they want to collectively work out whether they can choose a hospital for each of the injured people in such a way that the load on the hospital is balanced: each hospital receives at most n/k people. Give a polynomial-time algorithm that takes the given information about the people's and the hospitals' location, and determines whether this is possible.

Solution:

In the figure below, there are n injured people denoted as P_i and k hospitals denoted by H_i. Not every hospital is accessible to every patient. So there can be atmost k edges from a patient to hospitals.

To determine if each patient can be mapped to a hospital without overloading the hospitals (atmost n/k in every hospital), we can find out the flow of this network. The flow out of sink is n (n patients with weight 1 for each) and the flow in sink is $k.\ (n/k)$ i.e n (n/k for every hospital and k such hospitals). If the maximum flow of this network is less than n that means atleast one of the patient is not mapped to any of the hospitals.

Using Ford–Fulkerson algorithm, calculate the maximum flow of this network and if this value less than n then it is not possible to map every patient to a hospital (with constraint of n/k patients per hospital and only hospitals within half hour drive)



Q8

Here's a problem that occurs in automatic program analysis. For a set of variables x1, …, xn, you are given some equality constraints, of the form "xi=xj" and some disequality constraints, of the form "xi¹xj". Is it possible to satisfy all of them? For instance, the constraints x1=x2, x2=x3, x3=x4, and x1¹x4 cannot be satisfied. Devise an efficient algorithm that takes m constraints over n variables and decides whether the constraints can be satisfied.

Solution:

The problem can be modelled such that all equality variables are assigned to one set. If at any point there is a, conflict, it means that the constraints cannot be satisfied.

Pseudo

1. Start a BFS traversal from any node s. Assign this variable a set number (say S_1)
2. While visiting the neighbors of s, based on the equality sign assign them sets. For example, if $s = u$, assign u to S_1. If $s! = v$, assign v to S_2.
3. Before assigning any node to a set check if the node was present in any of the two sets before. If no, assign it to the intended set. If yes, check if the set it was found in is same as the intended set. If there is a conflict, return "Constraints not satisfied".
4. Repeat this step for all the nodes.

If this program doesn't return "Constraints not satisfied", all the constraints are satisfied.