AAD - Assign2 - Problem Set 2

Q1) Given Graph - G, n-vertices, m-edges, distinct edge weights

edge - e

Approach 1:-

As all the edge weights are distinct => 1 MST

we can first build the MST then check if edge e is present in it (or) not

Let us use ① Kruskal's Algorithm for building the MST => O(mlogn) - time

② Then we get a MST with only n-1 edges and by using an array we can store these edges while applying kruskal's.

Now it takes O(n) time to search for edge e

if e is present => Then e is in MST

else => e is NOT in MST

Total time complexity :- O(mlogn + n)

Correctness :-

As the edge weights are distinct

=> There is only one MST possible

and we know that kruskal's Algorithm is correct hence MST is correct

thus checking the MST edges doesn't affect the correctness => The Algorithm is correct

Approach 2:

edges are distinct ⟹ only one MST

Let edge e be between vertices (v,w)

① Delete edge e from G and Start DFS/BFS from v to reach w on the condition "that we can only travel through edges with weight less than e"

② If we can reach w then e ∉ MST else it belongs to MST.

we can reach w only if

① ∃ a cycle containing e in it (another path for v→w)

② e is the most expensive edge in that cycle as we only traverse through edges lighter than e

Runtime: O(|E|) = for a single DFS/BFS

correctness:

W.K.T in reverse-delete algorithm

"Given edge weights are distinct. If there is a cycle C in Graph G and let e(v,w) be most expensive edge in C then MST doesn't contain e in it"

Hence the algorithm returns false when it is most expensive edge in the cycle

Thus it always gives correct output

Q2

Given, the Graph $G=(V,E)$ with edge costs $c_e \geq 0$
edges - $e \in E$

T = MST of G

and Assume that all c_e (edge weights) are distinct => T is unique and only one MST exists

- New edges be (v,w)

1. Algorithm:

- If (v,w) is in T then it has to be in G and if not - then we don't care whether it is there in G.

So we can test for new edge by directly adding it to T.

=> Add the edge (v,w) to T and form T'

Now as adding an edge to MST forms a cycle

=> T' has a cycle and the cycle contains the new edge (v,w)

- Now by ~~Cycle property~~ reverse delete - "We can delete highest weight edge from the Cycle to get MST"

- Here if edge (v, w) is highest weight edge then T remains MST

else T no longer remains MST
We delete the highest weight edge found from T' to get new MST
Hence obtaining updated T.

(2) Run Time:

For detecting the cycle and finding the highest weight edge we can run DFS/BFS from new edges

- checking: start DFS/BFS from v and go through edges $< c$ (cost of (v,w)) only to reach w

(v,w) is deleted - if w is reached then T remains MST
else T is no longer MST

during this call we can update highest value edge and delete it at the end

→ Time Complexity $= O(|E|) = \cancel{O(m)}$ only $n-1+1$ edges
$= O(n)$

(3) Correctness :-

The Algorithm returns/removes the most expensive edge from the cycle in T".

and w.k.T

" if all edges weights are distinct. and let Let c be any cycle in G(Graph) and let e(v,w) be most expensive edge in C. then MST doesn't contain e in it"

Hence removing the highest weight edge always results in correct MST.

Q5) Graph - $G(V, E)$ $d_1, d_2 \dots d_n$ - degree's

① **Algorithm:-**

- we first design a recursive function that takes the ~~sorted~~ list of degree's as an argument and
  - return's true if $G(V, E)$ exists
  - else return's false

② then we ~~first~~ sort all the degree's d_i's in non-increasing order

$\Rightarrow d_1 \geq d_2 \geq d_3 \geq \dots \geq d_n$

③ Now call the recursive function

$\text{recur}(\underbrace{\{d_1, d_2, \dots d_n\}}_{\text{size}})$

④ in the function

- check if first element (d_1) is 0
  - if yes return true
- ~~else else~~
- otherwise pop the first element (d_1)
- check if the size of list containing remaining elements is less than d_1
  - if yes then return false
- ~~otherwise con~~
- otherwise do $\boxed{d_i -= 1}$ $\forall i = 2$ to n

(subtract one from all d_i's)

- Now if any ($d_i < 0$) after updating then return false

- If the function doesn't return yet then call the $\boxed{\text{recur}(d_2, \sim d_n)}$

i.e., call again with the updated list

- we can send ~~in~~ start index as a parameter so that popping d_1 is $O(1)$

(2) **Run Time:**

- If there are n d_i's then at every recursive call we may sort in $O(n \log n)$ and we may do it for all n number's (d_i's)

$\Rightarrow O(n * n \log n)$

$O(n^2 \log n)$

- and for $\boxed{d_i - 1}$ operation $\forall i = 2$ to n applying this operation $\forall n$ for each recur() call

$\Rightarrow O(n * n)$

Total $= O(n * n) + O(n * n \log n)$

$= O(n^2 \log n)$

$= O(n^2 . \log n)$

Note: I have seen "Havel-Hakimi Theorem" to solve this problem

3) Correctness:

This is just same as "Havel - Hakimi Algorithm" which uses the theorem:

"The non-increasing sequence $(d_1, d_2 \ldots d_n)$ is graphic iff the sequence

$(d_2-1, d_3-1, \ldots, d_{d_1+1}-1, d_{d_1+2}, d_{d_1+3} \ldots d_n)$

is also graphic"

proof:

$(\Leftarrow)$ if $(d_2-1, d_3-1, \ldots, d_{d_1+1} \ldots d_n)$ is graphic

$\Rightarrow (d_1, d_2 \ldots d_n)$ is graphic

- we can just add a new vertex and add edges which connects it to all vertices corresponding to $(d_2-1, d_3-1, \ldots d_{d_1+1}) \ldots d_n)$

$(\Rightarrow)$ if $(d_1, d_2 \ldots d_n)$ is graphic

then $(d_2-1, d_3-1, \ldots d_{d_1+1}, \ldots d_n)$ is graphic

let v_i be the vertex with degree d_i

if v_1 doesn't have an edge from all $(v_2, \ldots v_k)$

let it be $v_j \Rightarrow (v_1, v_j)$ edge doesn't exist

but $j > k$ & (v_1, v_j) exists ~~for~~ $i < j$ $d_i \geq d_j$

$\Rightarrow v_i$ has an edge to v_k s.t. (v_i, v_k) but ~~for~~ (v_j, v_k) doesn't exist

then remove ~~(v_1, v_m)~~ $(v_1, v_j), (v_i, v_k)$ and

add $(v_1, v_i), (v_j, v_K)$ - no change in d_i's

still graph remains same as given in theorem

Hence proved

Q4

Given

Graph $G(V,E)$ - No. of nodes $= n$

- each pair of nodes are joined by an edges
- $w_{i,j}$ denotes the edgeweight of nodes (i,j) edge
- It holds ~~traing~~ triangle Inequality

$$\boxed{w_{i,k} \le w_{i,j} + w_{j,k}}$$

RTP: finding minimum-weight Steiner Tree on X takes $\boxed{n^{O(k)}}$ time

Now

$X \subseteq V$ X is set of terminals

(No. of Terminals $= k$)

$X \subseteq Z \subseteq V$

Z = set s/t $X \subseteq Z \subseteq V$

with a ~~MST~~ spanning SubTree T of $G[Z]$

The weight of Steiner Tree = Weight of Spanning Tree T.

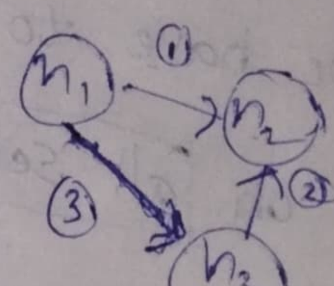
$|X| = k$ —— (1)

let set Z ~~have extr~~ be s/t

Steiner Tree T on $X \cup Z \subseteq V$

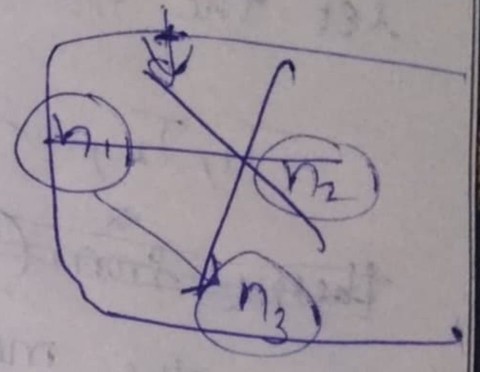
Now we see the degree of each node in T

(1) If degree of each node ≤ 2



then by Triangle Inequality

$W_{n_1,n_3} \leq W_{n_1,n_2} + W_{n_2,n_3}$



we can ~~replace the 2 incident edges~~ get minimum MST

(2) ~~a~~ Thus degree ≥ 3 in Steiner Tree T

Now

W.K.T

No. of leaves in $\geq$ No. of nodes with
a Tree degree ≥ 3

in this setup

Now

$|Z| \le \left(\begin{array}{l} \text{No. of Terminals/ leaves} \\ \text{whose degree} \ge 3 \end{array} \right)$

$\le$ Total No. of Terminals

$\le |X|$

$\le k$

Hence $\boxed{|Z| \le k}$

Now for minimum Steiner Tree

we have to calculate MST

on $\boxed{\text{all sets } X \cup Z \quad s/t \quad |Z| \le k}$

let the MST's formed by above operation be

$T_1, T_2, \ldots T_m$

~~then from (1) the minimum~~

then the minimum weighted Tree among all T_i's $i \in \{1, \ldots$

will ~~have~~ be minimum Steiner Tree (I (1))

$\Rightarrow$ the cheapest MST will be minimum Steiner Tree

Thus we have $^nC_{2k}$ options for all possible set formation's

$\therefore$ Overall Time Complexity $= {}^nC_{2k}$

$\boxed{= n^{O(k)}}$

Hence proved

Note: I took help of Text book - (Klienberg & Tardos) for solving this problem.