AAD End-semester Exam Rubric

QUES 1) There are several methods to solve this question, as it is a relatively easy question.
One popular method is using the sum of degrees of the graph nodes. This is approximately the
proof:
Let x = |U|. Also n = |V|
So, the number of nodes with degree 1 or 2 = n- x
Now, let k be the sum of degrees of nodes in U and let m be the sum of degrees of nodes in
V \ U
k >= 3x and m>=n-x(As the minimum degree in the two sets is 3 and 1, respectively)
Now, for a tree, the number of edges is n- 1, and total sum of degrees is 2(n- 1)
Therefore, 2(n- 1) = k + m >= 3x + (n- x)
⇒ n- 2 >=2x
⇒ x <= n/2- 1
⇒ x <= n/2
⇒ |U| <= |V| / 2
Hence proved.
There are many different ways to show this inequality. Any approach that gives the correct
inequalities step-by-step can get full marks.
Another approach involves showing that the number of nodes with degree 1 >= the number of
nodes with degrees greater than or equal to 3.
For this, one could also use either the sum of degrees or Induction.

Marking Scheme:
Here is a rough outline, as people have come up with the inequalities in diverse ways:
You can get 1 mark for showing the inequality that the sum of degrees of nodes in U >= 3|U|

You get another 1 mark for calculating the sum of degrees of the remaining nodes.
You get 1 mark for pointing out that the number of edges in a tree is n- 1 and that the sum of
degrees is equal to 2(n- 1).
You get 1 mark for setting up the final inequality and another 1 mark for solving it correctly to
get the answer.
Anyone who has done a similar thing would get marks around approximately based on this key
or overall correctness of their approach, in case it is very different from this solution.

QUES 2)

Approach 1:

Case i) u and v belong to a strongly cc in G → then they belong to a strongly cc in G_{rev}

Since u and v belong to a SCC in G, then by definition, a path from u to v exists and similarly a path from v to u exists in G.

A u to v path in G becomes a v to u path in G_{rev} and similarly a v to u path in G becomes a u to v path in G_{rev} and hence u and v also belong to the same SCC in G_{rev}.

Case ii) u and v belong to different SCCs in G → then they belong to different SCCs in G_{rev}

Since u and v belong to different SCCs in G then either u to v path does not exist or v to u path does not exist in G. WLOG assume that u to v path does not exist in G. Then in G_{rev} u to v path will not be there as only edge direction is reversed and no new edges are created.

Hence, u and v will belong to different SCCs in G_{rev} also.

Approach 2: Prove that a SCC in G remains in a SCC in G_{rev} and also a SCC in G_{rev} remains in the same SCC in G.

Let's say u and v belong to a SCC in G. This means that there exists a path from u to v and a path from v to u in G. Upon changing the edge directions we can say

that the u to v path in G becomes v to u path in G_{rev}. Similarly, the v to u path in G becomes a u to v path in G_{rev}. Thus, u and v belong to the same SCC in G_{rev} as well.

Using the same argument we can say that two vertices belonging to a SCC in G_{rev} will be in the same SCC in G as$(G_{rev})_{rev}$ = G

Approach 3 (proof by contradiction):

Assume G and G_{rev} do not have the same strongly connected components. WLOG, assume that there exists a SCC in G (let's say C) and all vertices of C are in components $C_1, C_2, .., C_k$ in G_{rev}.

Let u and v belong to C in G that belongs to different components $u \in C_1$ and $v \in C_2$ in G_{rev}. This means that there is no path from u to v OR from v to u in G_{rev}. But since these vertices belong to the same SCC C in G, a path from u to v and v to u exists in C and should also exist in G_{rev} as only the edge directions are reversed. Hence, a contradiction.

Approach 4 (another proof by contradiction)

This requires you to prove contradiction for all cases:

Case i) Assume u and v belong to same SCC in G then u and v belong to different SCC in G_{rev}. We will prove that this is a contradiction.

Since u and v belong to same SCC in G, this means that there exists a path from u to v and a path from v to u in G. Upon changing the edge directions we can say that the u to v path in G becomes v to u path in G_{rev}. Similarly, the v to u path in G becomes a u to v path in G_{rev}. Thus, u and v belong to the same SCC in G_{rev} as well. Hence, a contradiction.

Case ii) Assume u and v belong to different SCCs in G then they belong to same SCC in G_{rev}. We will show that this is a contradiction.

Since u and v belong to different SCC in G, assume that $u \in S_1$ and $v \in S_2$ where S_1 and S_2 are different SCCs in G. Then either a path from u to v does not either or a path from v to u does not exist or no path exists between them(i.e. disjoint SCCs). Easy to say for disjoint. WLOG, assume that u to v path exists but v to u path does not exist. Then, upon edge reversal, u to v path becomes a path from v

to u and since reversal does not generate any new edges, no path from u to v exists. Hence, u and v cannot belong to same SCC in G_{rev}. Hence, a contradiction.

Marking Scheme:

→ Full marks(5) for any of the above three approaches.

→ Partial marks depending on how correct they were. For ex, proving only one direction in approach 2 or 4 would fetch 2.5 marks.

→ If completely wrong proof, then grace mark of 1 if correct property of a SCC is stated.

Some common mistakes for which marks have been cut:

→ Algo for Kosaraju shows that reversing the edge directions doesn't affect the SCCs, hence G and G_{rev} have same SCCs.

→ Reverse DFS gives same SCCs hence G and G_{rev} have same SCCs.

→ Apply Kosaraju in G and G_{rev} and both will give same SCCs hence G and G_{rev} have same SCCs.

→ "If u and v are reachable in G then they will also be reachable in G_{rev}" (not shown that u to v path in G becomes v to u path in G_{rev} and vice-versa due to simple edge reversal)

→ Given an example and reversed edges and said that it is same SCC and hence G and G_{rev} will also have same SCC.

→ Used word *edge* instead of *path* (2.5 marks cut) {For ex: *edge* from u to v is reversed in G_{rev} and becomes *edge* v to u}

→ Written "u→v" and "v→u" without saying anything if that is path or edge (2.5 marks cut)

→ Mentioned that if u and v belong to different SCC in G and a path from u to v exist then a path from v to u will exist in G_{rev}. (This is incomplete as if u and v belong to belong to different SCC in G then either u to v path will exist or v to u path will exist or no path will exist)

→ Topological sort of G and G_{rev} gives same SCC hence, they have same SCC.

→ “Reversing edges won’t change SCC and hence G and G_{rev} have same SCC” (that’s what you have to prove)

QUES 3)

Correct binary search (4 marks):

- if a[mid] == mid: return true
- else if a[mid] > mid: go to left half
- else go to right half

Proof of correctness (4 marks):
Claim: If a[mid] > mid, then a[i] > i for all i > mid.
Proof:

a[i] >= a[mid] + i - mid
⇒ a[i] > mid + i - mid = i

Pruning to the other side can be shown similarly. Any argument that conveys this idea properly is given credit.

QUES 4)

Part a)

Assume MST is not a MBST.

Let T_1 be the MST of the Tree, and T_2 be a MBST(spanning tree with a lighter bottleneck edge).

Then T_1 has an edge e_1 which has a heavier edge weight than all edges in T_2 as MST isn't the MBST.

Now if we add the edge e_1 to T_2, it forms a cycle C on which it is the heaviest edge (since all other edges in C belong to T_2).

So by the cycle property, e_1 does not belong to any minimum spanning tree, contradicting that it belongs to T_1. Thus we arrive at a contradiction.

Marking Scheme:

[1 mark] is for defining the two trees and tell that MST has an edge e_1 not in MBST

[1 mark] for stating that e_1 has a weight larger than any edge in MBST.

[1.5 mark] for adding this edge e_1 to MBST and saying that a cycle is formed with e_1 being largest.

[1.5 mark] for using cycle property+ to say that e_1 should not be in MST. So contradiction.

Also give 3 marks if used cut property correctly instead of cycle property.

→ Any other valid proof that captures the idea well will be considered.

Part b)

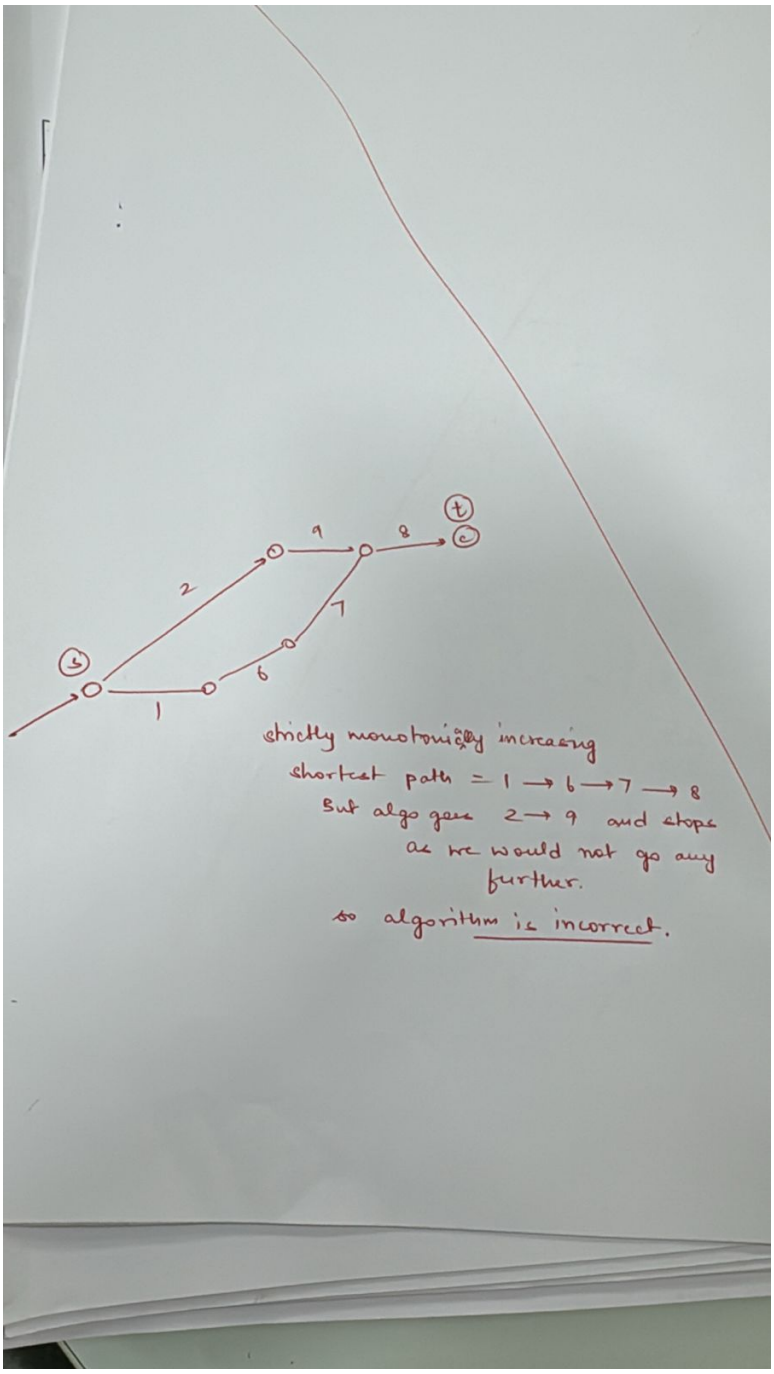
Just don't take the lowest edge in a cycle, as long as this edge is still less than the bottleneck. Then you have a MBST which is not an MST.

Marking Scheme:

[1 mark] No.

[2 mark] Correct counter example.

QUES 5) part a)



part b)

Dijkstra mentioned + Monotonicity proved: 3 Marks.
Proved that all valid paths of G are in G' as well: 3 Marks.

QUES 6)

The correct answer is YES and the proof that running the Kruskal algorithm works in this case is given below -

Proof. Let $\mathcal{C}$ denote the clustering $C_1, C_2, \ldots, C_k$. The spacing of $\mathcal{C}$ is precisely the length d^* of the $(k-1)^{\text{st}}$ most expensive edge in the minimum spanning tree; this is the length of the edge that Kruskal's Algorithm would have added next, at the moment we stopped it.

Now consider some other k-clustering $\mathcal{C}'$, which partitions U into non-empty sets $C'_1, C'_2, \ldots, C'_k$. We must show that the spacing of $\mathcal{C}'$ is at most d^*.

Since the two clusterings $\mathcal{C}$ and $\mathcal{C}'$ are not the same, it must be that one of our clusters C_r is not a subset of any of the k sets C'_s in $\mathcal{C}'$. Hence there are points $p_i, p_j \in C_r$ that belong to different clusters in $\mathcal{C}'$—say, $p_i \in C'_s$ and $p_j \in C'_t \neq C'_s$.

Now consider the picture in Figure 4.15. Since p_i and p_j belong to the same component C_r, it must be that Kruskal's Algorithm added all the edges of a p_i-p_j path P before we stopped it. In particular, this means that each edge on

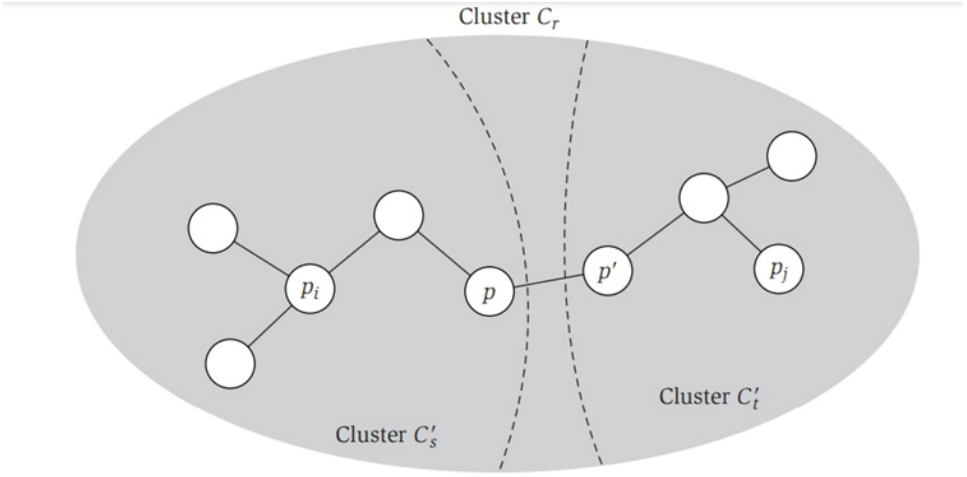


Figure 4.15 An illustration of the proof of (4.26), showing that the spacing of any other clustering can be no larger than that of the clustering found by the single-linkage algorithm.

P has length at most d^*. Now, we know that $p_i \in C'_s$ but $p_j \notin C'_s$; so let p' be the first node on P that does not belong to C'_s, and let p be the node on P that comes just before p'. We have just argued that $d(p, p') \leq d^*$, since the edge (p, p') was added by Kruskal's Algorithm. But p and p' belong to different sets in the clustering $\mathcal{C}'$, and hence the spacing of $\mathcal{C}'$ is at most $d(p, p') \leq d^*$. This completes the proof. ■

Marking scheme -

1. If written “No” <= 1 marks
2. Given the whole correct proof with proper explanation – 10 marks
3. Time complexity analysis carries 2 marks – 1 mark for writing (ElogE) and 1 mark for mentioning E = nC2 or n^2.
   (If written n^2 log (n^2) ⇒ n^2 log(n) then also given 2 marks).
4. Algorithm was already given in the question so if algorithm written (explained how kruskal works) but not mentioned the correct proof then 2 marks are given (as grace).

Common answers and marks provided –

1. Induction proof
   Answer written –
   Assume that the algorithm works for the k+1 clusters and then we have to add one edge to get the k clusters i.e. join two clusters. Argument is given that we

have edges joining these k+1 clusters and we choose the minimum edge among these so that it is not counted in the spacing and hence we are able to obtain the maximum spacing.

Flaw –
The given proof is only correct when proof about the following thing is also provided alongside -
“A way to obtain the optimal clustering for k cluster is by choosing the minimum edge in the k+1 cluster optimal arrangement.”
This is because it is not at all intuitive that we just add the minimum edge in the k+1 clusters since it can be possible that we break some clusters into components and then make some connections to obtain the optimal arrangement for the k cluster case.

Writing the simple induction as above without the required proof gets 3-4 marks since joining the minimum edge is simply what the algorithm says and there is nothing new to it.

2. Direct Argument.
   Answer written -
   We make use of the MST and a new edge connects two components and merges them into one single component.
   So if we do this until we get k disconnected components all the minimum possible edges have been already taken so the only edges left are the maximum weight edges and hence the spacing is maximised.

Flaw –
Most of the batch has done this. These is not a proof. You have stated the method given in problem and then simply claimed it will give the correct answer without proving it.
Consider the simplified problem – Alice has an array of integers 1 to n. She can remove x (<n) numbers in such a way that she wants to maximise the smallest integer left AFTER all x removals. To do this, Alice removes the x smallest integers. Prove that her method is correct.
Proof to this simplified problem is NOT “This method is correct because removing smallest integers will maximise the smallest integer”. This is NOT a proof as you have just stated the method again. Do attempt a correct proof of this problem.

Answer written –
We stop the clustering process before it add its last k-1 edges because at that point all vertices will be connected to its close by vertices and adding an edge would add an edge between 2 clusters combining them into one.
Since the only remaining edge weights are the largest k-1 edge weights we have ensured that the we have the maximum spacing.
Flaw -
Similar to above.

QUES 7)

1 Solution

Note: no two lines are parallel and no three lines intersect at the same point.

1.1 Algorithm

First, sort the lines in order of increasing slope. (1 mark)

1.1.1 Base Case

If $n \leq 3$, we can find the visible lines in constant time.

- *if* $n \leq 2$ return the lines (1 mark)
- *if* n *is* 3
  - the first and last lines will be visible (1 mark)
  - second line meets the first line to the left of where third line meets the first line $\Leftrightarrow$ second line is visible (1 mark)
  - return the visible lines

1.1.2 Recursive Case

We divide the lines at $m = \lceil \frac{n}{2} \rceil$, recursively computing the visible lines among $\{L_1, \cdots, L_m\}$ and $\{L_{m+1}, \cdots, L_n\}$.

Also return the sequence of values $a = \{a_1, \cdots, a_{p-1}\}$ where a_k is the points of intersection of L_{i_k} and $L_{i_{k+1}}$ for visible lines $\mathcal{L}_l = \{L_{i_1}, \cdots, L_{i_p}\}$, as well as b for $\mathcal{L}_r = \{L_{j_1}, \cdots, L_{j_q}\}$.

These points will be in order of increasing x coordinate.

$$a, \mathcal{L}_l = getVisibleLines(\{L_1, \cdots, L_m\})$$
$$b, \mathcal{L}_r = getVisibleLines(\{L_{m+1}, \cdots, L_n\})$$

(1 mark)

1.1.3 Merge

We merge the sorted lists a and b into a single list of points $c_1, c_2, \cdots, c_{p+q-2}$ ordered by increasing x coordinates. (1 mark)

For each $k \in [1, p+q-2]$, we check the line that is uppermost in $\mathcal{L}_l$ and $\mathcal{L}_r$ at c_k. Let s be the smallest index for which $uppermost(\mathcal{L}_r)$ lies above $uppermost(\mathcal{L}_l)$ at c_s.

Let the two lines at this point be $L_{i_d} \in \mathcal{L}_l$ and $L_{j_f} \in \mathcal{L}_r$, and let (x', y') denote the point of intersection of the two lines. Thus, $x' \in (c_{l-1}, c_l)$.

$\Rightarrow L_{i_d}$ is uppermost in $\mathcal{L}_l \cup \mathcal{L}_r$ to the left of x' and L_{j_f} is uppermost in $\mathcal{L}_l \cup \mathcal{L}_r$ to the right of x'.

(1 mark)

Thus, return the set of visible lines as $L_{i_1}, \cdots, L_{i_d}, L_{j_f}, \cdots, L_{j_q}$ and the set of intersection points is $a_{i_1}, \cdots, a_{i_{d-1}}, (x', y'), b_{j_f}, \cdots, b_{j_{q-1}}$.

Identify that the lines in the left set before the intersection point and the lines in the right set after the intersection point are to be returned (and corresponding points). (1 mark)

Identify that the intersecting lines and intersection point are to be returned along with the above. (1 mark)

1.2 Proof of Correctness

We prove the algorithm by proving some non-trivial aspects of each sub-step above. The correctness of the $getVisibleLines$ call is asserted through the correctness of the base case, recursive case, and merge step.

The algorithm largely depends on the observation that if two lines are visible, then the region in which the line of smaller slope is uppermost lies to the left of the region in which the line of larger slope is uppermost, else the line of smaller slope would be 'covered' by the line of larger slope.

Overall, for a valid proof: (2 marks)

1.2.1 Recursive Case

The reason the intersection points are in order of increasing x coordinate is from the observation above.

1.2.2 Merge Step

$x' \in (c_{l-1}, c_l)$ because L_{i_d} is over L_{j_f} at c_{l-1} but under it at c_l.

1.3 Time Complexity

The initial sort call is $O(nlog(n))$.

We split our lines into half and recursively call the algorithm on each set in each call. Merging the intermediate results takes $O(n)$ as we first merge two sorted arrays and then iterate over them with each loop operation being $O(1)$.

$$T(n) = 2T(\frac{n}{2}) + O(n) \rightarrow O(nlogn)$$

$$\text{Overall Complexity: } O(nlogn) + O(nlogn) = O(nlogn)$$

$$(1 \text{ mark})$$

QUES 8)

Subpart (a): (4 Marks)

- n = Length of String a, m = Length of String b
- **Time Complexity:**
  - The algorithm uses a nested loop structure to fill a (n+1)×(m+1) DP table
  - At each cell , the algorithm calculates the minimum cost using up to **five operations** (deletion, insertion, substitution, match, and transposition).
  - The time complexity is **O(n × m)** since each cell requires constant time computation (O(1)).
- **Space Complexity:**
  - The DP table requires (n+1)×(m+1) memory to store distances for all substrings of a and b.
  - Thus, the space complexity is **O(n × m)** for the table.

Marks Distribution:

1. Correct identification of time complexity as **O(n × m)** – **2 Marks**
   a. 0.5 marks for correct time complexity
   b. 1.5 marks for correct explanation
2. Correct identification of space complexity as **O(n × m)** – 2 **Marks**
   a. 0.5 marks for correct space complexity
   b. 1.5 marks for correct explanation

Subpart (b): (6 Marks)

```
#include <iostream>
#include <vector>
#include <queue>
#include <string>
using namespace std;

int main()
{
  string a, b;
  cin >> a >> b;
  int n = a.length();
  int m = b.length();
  int dp[n + 1][m + 1];

  for(int i = 0; i <= n; i++)
  {
    for(int j = 0; j <= m; j++)
    {
      dp[i][j] = 1e6;
    }
  }

  for(int i = 0; i <= n; i++)
  {
    dp[i][0] = i;
  }

  for(int i = 0; i <= m; i++)
  {
    dp[0][i] = i;
  }

  for(int i = 1; i <= n; i++)
  {
    for(int j = 1; j <= m; j++)
    {
      dp[i][j] = min(dp[i][j], dp[i - 1][j] + 1);
      dp[i][j] = min(dp[i][j], dp[i][j - 1] + 1);
      dp[i][j] = min(dp[i][j], dp[i - 1][j - 1] + (a[i - 1] == b[j - 1] ? 0 : 1));
```

```
      if(i > 1 and j > 1 and a[i - 1] == b[j - 2] and a[i - 2] == b[j - 1])
      {
        dp[i][j] = min(dp[i][j], dp[i - 2][j - 2] + (a[i - 1] == b[j - 1] ? 0 : 1));
      }
    }
  }

  for(int i = 0; i <= n; i++)
  {
    for(int j = 0; j <= m; j++)
    {
      cout << dp[i][j] << "    ";
    }
    cout << endl;
  }

  return 0;
}
```

DP Matrix:

	0	S	a	t	u	r	d	a	y
0	0	1	2	3	4	5	6	7	8
S	1	0	1	2	3	4	5	6	7
u	2	1	1	2	2	3	4	5	6
n	3	2	2	2	3	3	4	5	6
d	4	3	3	3	3	4	3	4	5
a	5	4	3	4	4	4	4	3	4
y	6	5	4	4	5	5	5	4	3

Marks Distribution:

0.125 each for every correct table entry except the ones mentioned in question paper (0.125 * 48 = 6 marks)

Subpart (c): (4 Marks)

Correctness of the Algorithm

The provided algorithm is incorrect. Any failing test case suffices to prove it incorrect.

Sample Failing Test Cases:

1. abcd bdac should give output as 3 but algorithm gives 4
2. from and mf should give output as 3 but algorithm gives 4

Marks Distribution:

1. Case “Correct”: 0 marks
2. Case “Incorrect”:
    a. 1 mark for writing that algorithm is “Incorrect”
    b. 3 marks failing test case / relevant explanation

QUES 9)

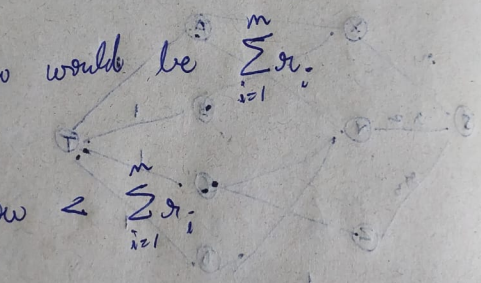
a) Outgoing edges will go to (courses) with respective (r_i) capacities

b) Incoming edges will come from (TA t_i) with 1 as the capacity for each.

c) In b/w, it can be any integer greater than or equal to 1

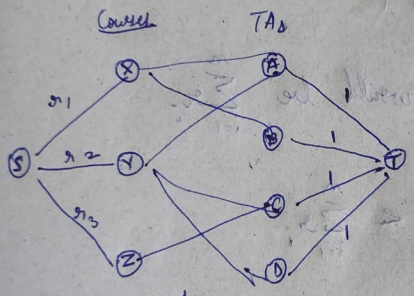
d) Max flow would be $\sum_{i=1}^{n} r_i$

e) Max flow $\leq \sum_{i=1}^{n} r_i$



Sir has given that edges are directed from course to TA (so while you could construct the flow differently, it is up to you to cut marks if they do TA to course instead)

The flow graph would be something like:



These edge weights can be any integer. But they must be atleast 1

If it is not feasible, max flow = min cut $< \sum x_i = R$

Let U be the set of nodes reachable from s in a min s-t cut.

Let C be the set of courses in U.

Clearly, $\sum_{i \notin C} x_i < R$ $\left(\because \sum_{i \in C} x_i \leq \text{min-cut} < R\right)$

Let T be union of sets of neighbours of nodes in C.

For each TA in T, it must either be disconnected from t } $= 1$

OR

disconnected from all of its neighbour nodes in C. } ≥ 1

$\therefore$ disconnecting all nodes in T from t is always optimal } total cost $|T|$

$\Rightarrow$ size of min-cut $= \left(\sum_{i \notin C} x_i\right) + |T| < R$

$\Rightarrow |T| < R - \sum_{i \notin C} x_i$

$\Rightarrow |T| < \sum_{i \in C} x_i$

$\therefore$ We found a set of courses C s.t. total number of TAs suitable for atleast 1 course in $C < \sum_{i \in C} x_i$