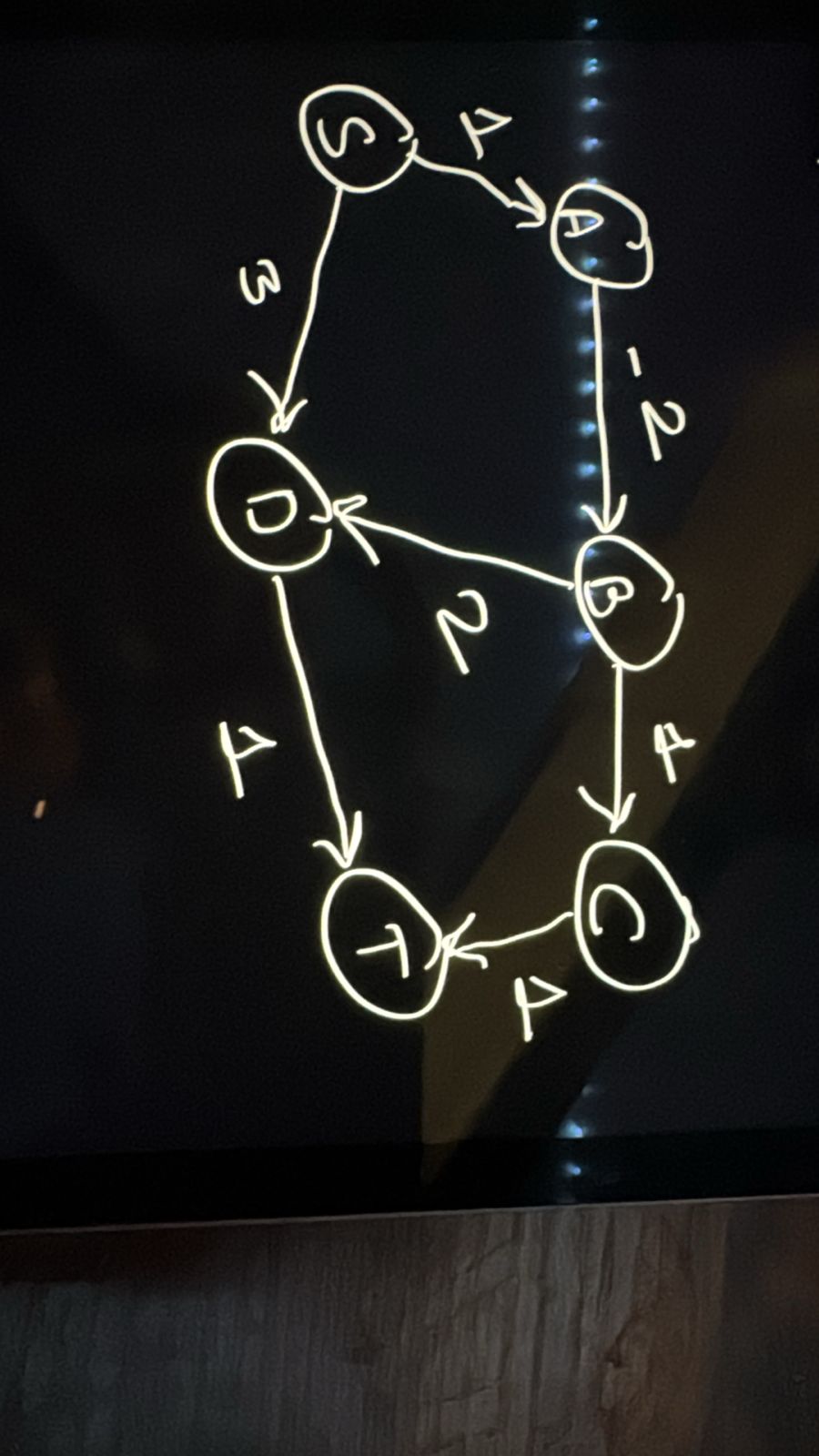
Name – Vipul Viresh Patil

Nuid – 002833236

Question 1:

(a) 

(b) **Increasing Edge Capacity and Maximum Flow**

If we increase the capacity of an edge by k units, we can potentially increase the maximum flow by at most k units. To find the new maximum flow, we can use the Ford-Fulkerson algorithm to find an augmenting path with a bottleneck capacity of at most k. This can be done in O(k|E|) time, where |E| is the number of edges in the graph.

(c) **False**: A local minimum in linear programming is **not** guaranteed to be a global minimum. This is because the feasible region of a linear program is a convex polytope, and there can be multiple local minima.

To find the global minimum, we need to use algorithms like the simplex method or interior point methods that explore the entire feasible region.

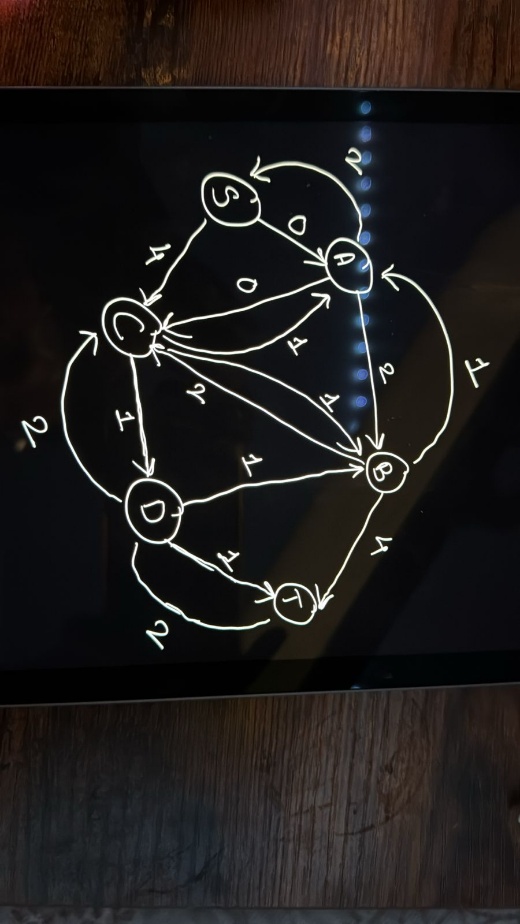
(d) **True.** If a cycle has a unique lightest edge, removing any other edge from the cycle will still leave the graph connected. Therefore, the unique lightest edge must be part of every MST, as removing it would disconnect the graph.

(e) **False.** Kruskal's algorithm is designed to find the minimum spanning tree, not the maximum spanning tree. To find the maximum spanning tree, we can modify Kruskal's algorithm to sort the edges in descending order of weight.

(f) **False.** Prim's algorithm, like Dijkstra's algorithm, relies on the assumption of non-negative edge weights. It may not work correctly if there are negative edge weights.

(g) The net flow across a cut (S, T) is equal to the value of the flow f. This is a fundamental property of flows in networks.

**Question 2:**

(a)Bottle Neck in first A -> C with capacity 1 and in second augmentation bottle neck is S -> C is 1. 

(b) All the augmenting path that could be chosen for the third augmentation step are

1. S -> C -> D -> T

2. S -> C -> D -> B -> T

(c) The numerical value of the maximum flow when the Ford-Fulkerson algorithm terminates is 3

(d) **Minimum Cut:**

The minimum cut separates the vertices into two sets:

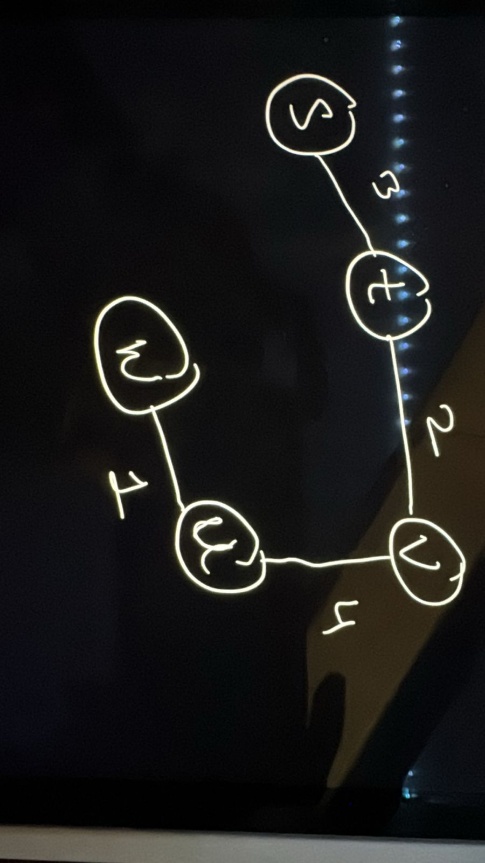
* **Set containing S:** {S, A, B}
* **Set containing T:** {C, D, T}

**Question 3:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Iteration | PQ | I(S) | I(A) | I(B) | I(C) | I(D) |
| 0 | [S] | 0 | ∞ | ∞ | ∞ | ∞ |
| 1 | [B, A] | 0 | -2 | -5 | ∞ | ∞ |
| 2 | [A, D] | 0 | -2 | -5 | ∞ | -4 |
| 3 | [D,C ] | 0 | -2 | -5 | 2 | -4 |
| 4 | [C] | 0 | -2 | -5 | -1 | -4 |
| 5 | [] | 0 | -2 | -5 | -1 | -4 |

**Question 4:**

(a)



**Order of Edges Added**:

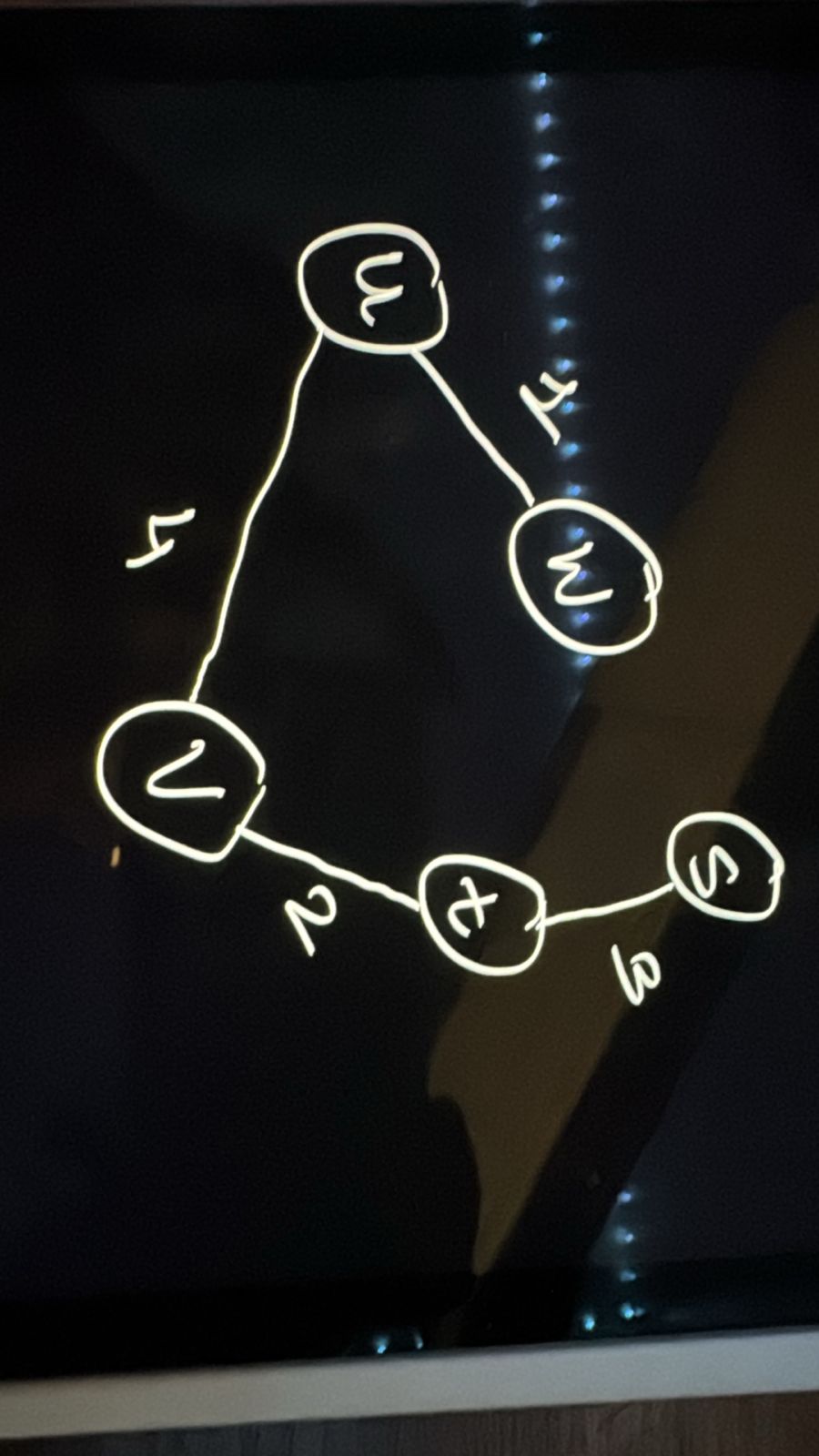
(s, t) weight 3

(t, v) weight 2

(v, u) weight 4

(u, w) weight 1

(b)



Order of edges Added

(u, w) weight 1

(t, v) weight 2

(s,t) weight 3

(v, u) weight 4

(c) **Cost of the Minimum Spanning Tree**

* The cost of the MST is the sum of all selected edge weights: 3+2+4+1=10 Both Prim’s and Kruskal’s algorithms result in the same MST with a total cost of **10**.

**Question 5:**

1. For the two jobs given: {t1 = 3, d1 = 1} and {t2 = 2, d2 = 3}

The two possible schedules are:

Schedule 1:

* Job 1 (t1 = 3) starts at T = 0, finishes at T = 3. Lateness l1 = max(0, 3 - 1) = 2.
* Job 2 (t2 = 2) starts at T = 3, finishes at T = 5. Lateness l2 = max(0, 5 - 3) = 2.
* Maximum lateness L = max(l1, l2) = 2.

Schedule 2:

* Job 2 (t2 = 2) starts at T = 0, finishes at T = 2. Lateness l2 = max(0, 2 - 3) = 0.
* Job 1 (t1 = 3) starts at T = 2, finishes at T = 5. Lateness l1 = max(0, 5 - 1) = 4.
* Maximum lateness L = max(l1, l2) = 4.

Therefore, the schedule with the lower maximum lateness is Schedule 1, with L = 2.

1. Counter-examples for the other two greedy choices:

(i) Shortest running time first: {t1 = 1, d1 = 2} and {t2 = 2, d2 = 1} This greedy choice leads to a maximum lateness of 1, whereas scheduling the jobs in earliest deadline first order results in a maximum lateness of 0.

(ii) Smallest slack first: {t1 = 2, d1 = 3} and {t2 = 1, d2 = 2} This greedy choice leads to a maximum lateness of 1, whereas scheduling the jobs in earliest deadline first order results in a maximum lateness of 0.

1. EDF(jobs):

sort jobs by increasing deadline

for job in sorted\_jobs:

start\_time = max(0, last\_finish\_time)

finish\_time = start\_time + job.t

lateness = max(0, finish\_time - job.d)

update maximum lateness

last\_finish\_time = finish\_time

Running time: O(n log n) for sorting, O(n) for the loop, so the total running time is

**O(n log n)**.

1. Sort the jobs in increasing order of deadline(dj):

* Job 5(d5 = 6)
* Job 3(d3 = 8)
* Job 4(d5 = 9)
* Job 2(d5 = 9)
* Job 6(d5 = 14)
* Job 1(d5 = 15)

Sorted jobs: [5, 3, 4, 2, 6, 1]

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Iteration | Job | Start | Finish | Lateness(max(0, finish - dj)) | Max lateness |
| 1 | 5 | 0 | 3 | Max(0, 3-6) = 0 | 0 |
| 2 | 3 | 3 | 5 | Max(0, 5-8) = 0 | 0 |
| 3 | 4 | 5 | 6 | Max(0, 6-9) = 0 | 0 |
| 4 | 2 | 6 | 10 | Max(0, 10-9) = 1 | 1 |
| 5 | 6 | 10 | 13 | Max(0, 13-14) = 0 | 1 |
| 6 | 1 | 13 | 15 | Max(0, 15-15) = 0 | 1 |

1. The idle time is the time duration between when the first job starts and when the last job finishes, minus the total runtime of all jobs.

In this case, the first job starts at 0 and the last job finishes at 15, with a total runtime of 2 + 4 + 2 + 1 + 3 + 3 = 15. Therefore, the idle time is 15 - 15 = 0.

Question 6:

* 1. The brute-force approach would be to generate all possible paths from s to t with at most k edges, calculate their lengths, and then return the shortest one. This can be done using a depth-first search (DFS) with a modification to limit the path length.

Running time : O(k\*m^k)

* 1. This problem is similar to the **Bellman-Ford algorithm**, where shortest paths are computed iteratively based on the number of edges.
  2. The key decisions we need to make to solve each subproblem are:
* Which node u to extend the path to from the current node v.
* Whether to update the shortest distance dist(v, i) using the new path
  1. dist(v, i) = min { dist(u, i-1) + l(u, v) } for all (u, v) in E
  2. **Base Cases:**
* dist(s, 0) = 0 (The distance from s to itself with 0 edges is 0.)
* dist(v, 0) = infinity for all v ≠ s (No path from s to v with 0 edges)
  1. **Running Time Analysis**
* Number of subproblems: |V| \* k (For each node v and each edge limit i)
* Time per subproblem: O(|E|) (Iterating over all edges to find the minimum)
* Total running time: O(|V| \* k \* |E|)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Dist(A, i) | Dist(B, i) | Dist(C, i) | Dist(D, i) | Dist(T, i) |
| i = 1 | 1 | inf | 2 | Inf | inf |
| i = 2 | 1 | 3 | 2 | 5 | Inf |
| i = 3 | 1 | 3 | 2 | 5 | 7 |

Question 7:

**Nodes:**

**Nodes:**

* Source node (s)
* Sink node (t)
* Patient nodes (P1, P2, P3, P4, P5)
* Hospital nodes (H1, H2, H3)

**Edges:**

* From the source node (s) to each patient node with capacity 1 (each patient can be assigned to one hospital).
* From each patient node to the hospital nodes that accept their insurance policy with capacity 1 (each patient can be assigned to one of the eligible hospitals).
* From each hospital node to the sink node (t) with capacity equal to the hospital's capacity (2 for H1 and H3, 3 for H2).

1. Linear Programming Formulation

Decision Variables:

x\_ij: Binary variable, 1 if patient P\_i is assigned to hospital H\_j, 0 otherwise.

Objective Function:

Maximize the total number of patients assigned to hospitals:

Maximize: Σ (Σ x\_ij) for all i, j

Constraints:

* 1. Patient Assignment Constraint: Each patient can be assigned to at most one hospital:

Σ x\_ij <= 1 for all i

* 1. Hospital Capacity Constraint: The number of patients assigned to each hospital cannot exceed its capacity:

Σ x\_ij <= H\_j for all j

where H\_j is the capacity of hospital H\_j.

* 1. Insurance Policy Constraint: A patient can only be assigned to a hospital that accepts their insurance:

x\_ij = 0 if hospital H\_j does not accept patient P\_i's insurance

Explanation of Notation:

x\_ij: This represents the decision variable for assigning patient i to hospital j.

Σ: This symbol represents the summation operator.

H\_j: This represents the capacity of hospital j.