Question 1 : Flow Networks and Vertex Connectivity

Part 1: Flow Network with Vertex Capacities (7 Marks)

Problem

We have a flow network (like a system of pipes) where not only the pipes (edges) have a capacity, but the junctions (vertices) also have a limit $\ell(v)$ on how much can flow through them. We need to show how to build a new, equivalent network that only has pipe capacities, but still respects the original junction limits.

Solution

We replace each limited vertex v with two new vertices, v_in and v_out, and an edge between them.

1. **Split the Vertex**: For every vertex v (except the main source s or sink t), we create two new vertices: v_in (the "entrance") and v_out (the "exit").

2. **Add Internal Edge**: We connect these two new vertices with a single, new edge (v_in, v_out).

3. **Assign Capacities**:
   - We set the capacity of this new edge (v_in, v_out) to be the original vertex limit: c'(v_in, v_out) = $\ell(v)$. This edge now models the vertex's bottleneck.
   - All original edges (u, v) that used to enter v are rerouted to enter v_in. They keep their original edge capacity, c(u, v).
   - All original edges (v, w) that used to leave v are rerouted to leave from v_out. They also keep their original capacity, c(v, w).

Proof of Correctness

We prove that the vertex-capacitated max-flow problem on G is equivalent to the standard edge-capacitated max-flow problem on the transformed graph G'.

Flow Preservation Construction. Let G = (V, E) be the original graph with vertex capacities $\ell(v)$ for all $v \in V \setminus \{s, t\}$ and edge capacities c(u, v). The transformed graph

G' = (V', E') is constructed by replacing every $v \in V \setminus \{s, t\}$ with two nodes v_in and v_out connected by an edge (v_in, v_out) of capacity

$c'(v_{in}, v_{out}) = \ell(v)$.

Every original edge $(u, v) \in E$ is replaced by an edge (u_out, v_in) in G' with the same capacity:

$c'(u_{out}, v_{in}) = c(u, v)$.

All incoming original edges to v now end at v_in and all outgoing edges start from v_out. The source s and sink t remain unchanged.

Claim 1: Every feasible flow in G corresponds to a feasible flow in G'. Let f be any feasible flow in G. Define a flow f' in G' as follows:

$f'(u_{out}, v_{in}) = f(u, v)$ for all $(u, v) \in E$,

$f'(v_{in}, v_{out}) = \Sigma_{(v,w) \in E} f(v, w)$,

where the equality follows from flow conservation at v in G. Since f satisfies edge capacities in G, we have

$f'(u_{out}, v_{in}) = f(u, v) \leq c(u, v) = c'(u_{out}, v_{in})$.

Also, because f satisfies the vertex-capacity constraint in G,

$f'(v_{in}, v_{out}) = \text{flow through } v \leq \ell(v) = c'(v_{in}, v_{out})$.

Flow conservation holds at every v_in and v_out in G' by construction, so f' is a feasible flow. Moreover, the total flow out of s in G equals that in G':

$|f| = |f'|$.

Claim 2: Every feasible flow in G' corresponds to a feasible flow in G. Let f' be any feasible flow in G'. Define a flow f in G by

$f(u, v) = f'(u_{out}, v_{in})$.

Since f' satisfies edge capacities in G', f satisfies edge capacities in G. Also, the flow through a vertex v in G equals f'(v_in, v_out) in G', and since

$f'(v_{in}, v_{out}) \leq c'(v_{in}, v_{out}) = \ell(v)$,

the vertex capacity in G is respected. Flow conservation holds in G because it holds at v_in and v_out in G'. The value of the flow is preserved:

$|f| = |f'|$.

Conclusion: Equality of Maximum Flow Values. From Claim 1 and Claim 2, there is a one-to-one correspondence between feasible flows in G and feasible flows in G' with identical flow value. Therefore,

max-flow(G) = max-flow(G').

Thus, the transformation correctly converts vertex capacities into edge capacities without changing the maximum flow value.

Marks Distribution (7 marks)

- **(1.5 marks)**: For correctly describing the transformation: splitting each vertex v into v_in and v_out.
- **(1.5 mark)**: For correctly assigning new capacities: c'(v_in, v_out) = ℓ(v).
- **(0.5 marks)**: For stating that all original incoming and outgoing edges of v are preserved by redirecting them to v_in and v_out respectively.
- **(0.5 marks)**: For mentioning that the source s and sink t remain unchanged in the transformed graph.
- **(3 marks)**: Proof of Correctness: Explaining why any flow through v must pass through (v_in, v_out) and is therefore bounded by ℓ(v), and why this ensures the maximum flow in the transformed graph equals the original vertex-capacitated max flow.

Part 2: Computing Vertex-Connectivity (3 Marks)

Problem

The vertex-connectivity k is the minimum number of vertices you must "delete" to disconnect the graph. Can we use our max-flow knowledge (including the vertex capacity trick from Part 1) to calculate this number?

Solution

Yes. We can find the connectivity between a pair of nodes (s, t) and then generalize. To find the s – t vertex connectivity:

1. **Build a Flow Network**: Take the original undirected graph G and create a directed graph G_s,t where each edge {u, v} becomes two directed edges, (u, v) and (v, u).

2. **Assign Capacities**: This is the key. We want to find the number of vertices in a cut, not edges.

- Set all edge capacities to infinity (c(e) = ∞).
- Set all vertex capacities to 1 (ℓ(v) = 1) for every vertex v that is not s or t.

3. **Solve**: This new network G_s,t has vertex capacities. We now apply our transformation from Part 1 to convert G_s,t into a standard flow network G'_s,t (where every v becomes v_in → v_out with capacity 1).

4. **Find Max Flow**: Compute the maximum s − t flow in G'_s,t. The overall graph connectivity k is the minimum of this value over all non-adjacent pairs (s, t).

Proof of Correctness (Why this works)

By setting all vertex capacities to 1, we are saying "only one unit of flow can pass through any vertex." By the max-flow min-cut theorem, the max-flow value is equal to the minimum cut. In our transformed graph G'_s,t, the only edges with finite capacity are the internal (v_in, v_out) edges, which all have capacity 1.

Therefore, the min-cut must be a set of these internal edges. The value of the max-flow will be equal to the minimum number of these "capacity 1" vertices that must be "removed" to stop all flow from s to t. This is the very definition of s − t vertex connectivity.

Marks Distribution (3 marks)

- **(1 mark)**: For describing the correct setup: setting edge capacities to ∞ and vertex capacities to 1.
- **(1 mark)**: For explicitly stating the method: apply the Part 1 transformation, then find the max-flow.
- **(1 mark)**: For the Proof of Correctness, explaining why the max-flow in this new graph equals the s − t vertex connectivity.

Important Notes

- **No marks will be granted in Part 2 without proper construction** of the transformed graph G'_s,t as described in Part 1.
- **The augmented path algorithm approach will not work for Part 2**, and no marks will be granted for solutions using only the augmented path algorithm.

Question 2 — Optimal Vertex Cover for a Tree

Part (a) — Minimum Vertex Cover on a Tree (6 marks)

Expected answer structure

- Describe the greedy / DFS-based idea.
- Give the algorithm explanation (post-order DFS).
- Provide a proof of correctness (split into two 1-mark parts: (i) all edges covered, (ii) cover is minimum).
- Justify why the algorithm runs in linear time.

Marking rubric (6 marks)

- **Algorithm idea & steps** — 3 marks
  - Correct use of rooting + post-order traversal and the local rule (include parent if any child is excluded). (3/3)
- **Proof of correctness** — 2 marks (split as below)
  - (i) **(1 mark)** Proof that every edge is covered by the produced set.
  - (ii) **(1 mark)** Proof that the produced set is *minimum* (optimal).
- **Time complexity** — 1 mark (linear-time justification).

Idea

Root the tree arbitrarily and do a post-order DFS. For each node `u`, after processing all children, include `u` in the vertex cover iff **some child `c` has not been included**. This bottom-up rule ensures every edge between a parent and child is covered, while avoiding unnecessary vertices.

Algorithm (3 marks)

1. Root the tree at any vertex `r`.
2. Initialise `inCover[v] = false` for all vertices `v`.
3. Perform a post-order DFS:
   - For node `u` (after visiting all children):
     - If ∃ child `c` with `inCover[c] = false`, set `inCover[u] = true`.
     - Else leave `inCover[u] = false`.
4. Output the set `C = { v | inCover[v] = true }`.

(Pseudocode optional — clear prose describing the traversal and rule is sufficient.)

Proof of correctness (2 marks)

(i) (1 mark) — All edges are covered.

Consider any edge `(u, v)` and suppose `u` is the parent and `v` the child. When `u` was processed in post-order, if `v` was not chosen (`inCover[v] = false`) then our rule forces `inCover[u] = true` , so `(u,v)` is covered by `u` . If `v` was chosen, the edge is covered by `v` . Hence every edge is covered.

(ii) (1 mark) — The cover is minimum.
Prove by induction on subtree size.

- Base: leaf nodes — no edges inside the subtree, so leaving leaf unselected is optimal.
- Inductive step: assume optimal covers for each child subtree. For subtree rooted at `u` :
  - If some child `c` is not selected by the algorithm, then the edge `(u,c)` must be covered, forcing selection of `u` in any valid cover (or selecting `c` which is not an option), so selecting `u` is essential.
  - If all children are selected, leaving `u` unselected is safe and strictly better than selecting `u` .
    Because subtrees are disjoint and choices don't conflict (tree has no cycles), these local optimal choices combine into a global optimum. Thus the algorithm returns a minimum vertex cover.

Time complexity (1 mark)

Each vertex is visited once in DFS; each edge considered only when handling its parent or child. Each node's decision is `O(1)` . So total time is `O(V + E)` — linear time.

Part (b) — Vertex Cover via Max-Flow / Matching (4 marks)

Important: No marks will be granted without construction.

Mark distribution:

- Construction – 1.5 marks
- Algorithm – 1 mark
- Proof of Correctness – 1 mark
- Time Complexity – 0.5 mark

1. Construction (1.5 marks)

- Color the tree bipartition by BFS (2-coloring): obtain sets `L` and `R` .

- Build a flow network `G'` with source `s` and sink `t`:
  - For every `u ∈ L`, add edge `(s, u)` with capacity `1`.
  - For every `v ∈ R`, add edge `(v, t)` with capacity `1`.
  - For every tree edge `(x, y)` with `x ∈ L`, `y ∈ R`, add edge `(x, y)` with capacity `1`.
- This is a unit-capacity bipartite network where maximum `s-t` flow equals the size of a maximum bipartite matching.

2. Algorithm (1 mark)

1. Run a max-flow algorithm (Dinic / Edmonds–Karp) on `G'` to compute maximum flow `f`.
2. From the residual graph, find set `S` of vertices reachable from `s`.
3. The minimum vertex cover is:

```
C = (L \ S) ∪ (R ∩ S)
```

3. Proof of Correctness (1 mark)

- **Validity:** Every tree edge connects `L` and `R`. The min-cut / reachable set construction yields a set `C` that intersects every edge, so `C` is a vertex cover.
- **Optimality / Approximation:** By Kőnig's theorem (for bipartite graphs), the size of a minimum vertex cover equals the size of a maximum matching (`|VC_min| = |M_max|`). Thus the min-cut from max-flow yields a minimum vertex cover. Selecting both endpoints of matched edges gives a valid 2-approximation.

4. Time Complexity (0.5 mark)

- Network construction: `O(V)`.
- Max-flow on a bipartite unit-capacity graph: `O(E √V)`; for a tree `E = V-1` → `O(V^{1.5})`.
- With specialized DFS-based augmentations for trees, the algorithm can be implemented in `O(V)`.

Question 3

1.1 Closure of P under Intersection (1 Mark)

Answer: Yes, the class P is closed under intersection.

Proof: Let L_1 and L_2 be two languages in P. By definition, there exist deterministic Turing machines (DTMs) M_1 and M_2 that decide them in polynomial time.

Let M_1 decide L_1 in time $O(n^{k_1})$ for some constant k_1, and let M_2 decide L_2 in time $O(n^{k_2})$ for some constant k_2.

We can construct a new DTM M_int to decide the language

$L = L_1 \cap L_2$.

On any input string w, the machine M_int operates as follows:

1. Run M_1 on w.
2. If M_1 rejects, then $w \notin L_1$, so reject.
3. If M_1 accepts, run M_2 on w.
4. If M_2 accepts, then accept; otherwise, reject.

Time Complexity Analysis: The total time taken by M_int is the sum of the times for M_1 and M_2:

$O(n^{k_1}) + O(n^{k_2}) = O(n^{\max(k_1,k_2)})$,

which is still polynomial.

Since M_int is a deterministic polynomial-time decider for $L_1 \cap L_2$, it follows that $L_1 \cap L_2 \in P$.

Hence, P is closed under intersection. □

1.2 Closure of NP under Intersection (2 Marks)

Answer: Yes, the class NP is closed under intersection.

Proof (using the Verifier definition): Let $L_1, L_2 \in NP$. By definition, there exist polynomial-time verifiers V_1 and V_2 such that:

$w \in L_1 \iff \exists c_1$ with $|c_1| = \text{poly}(|w|)$ and $V_1(w, c_1)$ accepts,

$w \in L_2 \iff \exists c_2$ with $|c_2| = \text{poly}(|w|)$ and $V_2(w, c_2)$ accepts.

We construct a verifier V_int for $L = L_1 \cap L_2$ as follows.

Certificate: Let the certificate for w be

c_int = (c_1, c_2),

where c_1 and c_2 are certificates for V_1 and V_2, respectively. Since both $|c_1|$ and $|c_2|$ are polynomial in $|w|$, their combined size $|c\_int| = |c_1| + |c_2|$ is also polynomial.

Verifier V_int: On input (w, c_int) where c_int = (c_1, c_2):

1. Run $V_1(w, c_1)$.
2. Run $V_2(w, c_2)$.
3. Accept if and only if both V_1 and V_2 accept.

Correctness and Time Complexity: The runtime of V_int is

$O(n^{k_1}) + O(n^{k_2}) = O(n^{\max(k_1,k_2)})$,

which is polynomial.

Soundness and Completeness:

- If $w \in L_1 \cap L_2$, then there exist c_1 and c_2 such that $V_1(w, c_1)$ and $V_2(w, c_2)$ both accept. Thus, V_int(w, (c_1, c_2)) accepts.
- If V_int(w, (c_1, c_2)) accepts, then both $V_1(w, c_1)$ and $V_2(w, c_2)$ accept, implying $w \in L_1$ and $w \in L_2$, so $w \in L_1 \cap L_2$.

Hence, there exists a polynomial-time verifier for $L_1 \cap L_2$, so $L_1 \cap L_2 \in$ NP. Therefore, NP is closed under intersection. □

Grading Rubric Summary

Part 1.1 §: 1 Mark total

- [0.25 marks] Correctly stating "Yes".
- [0.5 marks] Valid proof using two polynomial-time deciders.
- [0.25 marks] Showing the combined runtime is polynomial.

Part 1.2 (NP): 2 Marks total

- [0.5 marks] Correctly stating "Yes".
- [1.5 marks] Valid proof using verifier definition:
  - [0.5 marks] Defines NP via polynomial-time verifiers and certificates.
  - [0.5 marks] Constructs a combined certificate c_int = (c_1, c_2) and notes its size is polynomial.
  - [0.5 marks] Defines verifier V_int that runs both V_1 and V_2 and proves polynomial runtime.

2.1 Prove $CLIQUE_{1000} \in P$ (1 Mark)

Answer: To prove $CLIQUE_{1000} \in P$, we must provide a deterministic algorithm that decides it in polynomial time. The input is a graph G, and k = 1000 is a fixed constant, not part of the input.

Let n be the number of vertices in G.

Algorithm (Brute-Force):

1. Generate all possible subsets of 1000 vertices from G.
2. The total number of such subsets is (n choose 1000).
3. For each subset S:
    - 3.1. Check if S is a clique. This involves checking if an edge exists between all (1000 choose 2) pairs of vertices in S.
    - 3.2. If all edges exist, Accept and halt.
4. If no such subset is a clique, Reject.

Time Complexity Analysis:

- Checking one subset takes $O(1000^2)$ time. Since 1000 is constant, this is O(1).
- The number of subsets is (n choose 1000) = n(n−1)...(n−999)/1000! = O(n^1000).
- Total runtime: O(n^1000) × O(1) = O(n^1000).

Because k = 1000 is fixed, O(n^1000) is polynomial. Therefore, $CLIQUE_{1000} \in P$.

Grading:

- [0.5 marks] Describes the brute-force subset-checking algorithm.
- [0.5 marks] Correct complexity analysis (O(n^1000)) and explicitly states that it is polynomial since 1000 is constant.

2.2 Show CLIQUE_n/2 $\in$ NP-Complete (3 Marks)

Definition:

$$\text{CLIQUE\_n/2} = \{\langle G \rangle \mid \text{G has n vertices and a clique of size n/2}\}.$$

To prove this, we must show:

1. CLIQUE_n/2 $\in$ NP.

2. CLIQUE_n/2 is NP-hard.

1. CLIQUE_n/2 $\in$ NP (1 Mark)

We construct a polynomial-time verifier:

- **Input:** $\langle G \rangle$, with $n = |V(G)|$.
- **Certificate:** A subset $S \subseteq V(G)$.
- **Verifier V:**
  1. Check if $|S| \geq n/2$ (takes $O(n)$ time).
  2. Check if S is a clique: for all pairs in S, verify edges exist. ($O(|S|^2) = O(n^2)$ time.)
  3. Accept if both checks pass; otherwise reject.

The verifier runs in $O(n^2)$ time, hence polynomial. Therefore, CLIQUE_n/2 $\in$ NP.

2. CLIQUE_n/2 is NP-hard (2 Marks)

We reduce from the standard CLIQUE problem. Given $\langle G_0, k \rangle$ where G_0 has n_0 vertices, construct a graph G' such that:

$\langle G_0, k \rangle \in$ CLIQUE $\iff \langle G' \rangle \in$ CLIQUE_n/2.

Reduction Construction:

- **If $k \leq n_0/2$:** Add $m = n_0 - 2k$ new vertices forming a clique K_m, connected to all vertices in G_0. Then $G' = G_0 + K_m$, $n' = 2n_0 - 2k$, and the target size is $n'/2 = n_0 - k$.

  $\omega(G_0) \geq k \iff \omega(G') \geq n'/2$.

- **If $k > n_0/2$:** Add $m = 2k - n_0$ isolated vertices. Then $G' = G_0 \cup I_m$, $n' = 2k$, and the target size is $n'/2 = k$.

  $\omega(G_0) \geq k \iff \omega(G') \geq n'/2$.

This reduction is polynomial time, proving that CLIQUE_n/2 is NP-hard.

Grading:

- [1 mark] For the NP proof (verifier and polynomial analysis).
- [2 marks] For the NP-hard proof: [1 mark] for the general reduction idea; [1 mark] for the correct construction.

2.2 Alternate: Show that CLIQUE is NP-Complete

To show that CLIQUE is NP-Complete, we must satisfy two conditions:

1. CLIQUE $\in$ NP.
2. CLIQUE is NP-hard (i.e., every problem in NP can be reduced to it). We will prove this by showing that 3-SAT, a known NP-Complete problem, reduces to CLIQUE (3-SAT ≤_P CLIQUE).

Definition:

$\text{CLIQUE} = \overset{\text{G has a clique of size k}}{\langle G, \quad k \rangle}$,

where G is an undirected graph and k is a positive integer.

(1) CLIQUE $\in$ NP

A problem is in NP if a given solution (a "certificate") can be verified in polynomial time.

Instance: An input ⟨G, k⟩, where G is a graph and k is an integer.

Certificate: A set of vertices $S \subseteq V(G)$.

Verifier Algorithm:

1. Check if $|S| = k$.
2. For every pair of distinct vertices $(u, v) \in S$, check if the edge (u, v) exists in G.
3. Accept if both checks pass; otherwise reject.

Time Complexity:

- Checking size: O(n).
- Checking edges: $O(k^2) \leq O(n^2)$.

The total verification time is $O(n^2)$, which is polynomial. Hence, CLIQUE $\in$ NP.

(2) CLIQUE is NP-hard (Reduction from 3-SAT)

We show that 3-SAT ≤_P CLIQUE by constructing a polynomial-time reduction f.

Input (3-SAT): A 3-CNF formula

$\varphi = C_1 \wedge C_2 \wedge \cdots \wedge C\_m,$

where each clause C_i = (l_i,1 ∨ l_i,2 ∨ l_i,3) contains three literals.

Output (CLIQUE): An instance ⟨G, k⟩ such that φ is satisfiable ⟺ G has a k-clique.

Construction:

1. Set k = m (the number of clauses).
2. For each literal l_i,j in clause C_i, create a vertex v_i,j.
3. Add an edge between two vertices v_i,a and v_j,b iff:
    - a) They come from different clauses (i ≠ j), and
    - b) Their literals are not contradictory (i.e., l_i,a ≠ ¬l_j,b).

Correctness Proof:

(⇒) If φ is satisfiable, then there exists a truth assignment making each clause true. For each clause C_i, choose one literal l_i,a that is true under this assignment and select the corresponding vertex v_i,a.

- There is one vertex per clause, so $|S| = m = k$.
- Any two chosen literals are true simultaneously, so they cannot be negations of each other. Hence, all selected vertices are connected pairwise, forming a k-clique.

(⇐) If G has a k-clique S:

- By construction, no two vertices in S are from the same clause.
- Assign truth values such that every literal corresponding to a vertex in S is true.
- Because no two literals in S are contradictory, this assignment is consistent and makes at least one literal in every clause true.

Thus, φ is satisfiable.

Polynomial Time: The reduction creates 3m vertices and up to $O(m^2)$ edges, so it runs in polynomial time.

Conclusion:

φ is satisfiable ⟺ ⟨G, k⟩ ∈ CLIQUE.

Therefore, 3-SAT ≤_P CLIQUE. Since CLIQUE ∈ NP and is NP-hard, we conclude:

CLIQUE IS NP-COMPLETE.

Grading (3 Marks):

- [1 mark] Shows CLIQUE ∈ NP with a polynomial verifier.
- [2 marks] Shows CLIQUE is NP-hard via correct 3-SAT reduction:

- [1 mark] Correct construction (vertices, edges, k = m).
- [1 mark] Correct proof of equivalence (both directions).

2.3 Show CLIQUE_i+1 ≤_P CLIQUE_i (1 Mark)

Answer: We need to construct a polynomial-time (Karp) reduction.

Reduction (f):

1. Given G with vertices V = {v_1, . . . , v_n}.

2. For each v_j, construct the subgraph G_j = G[N(v_j)], induced by v_j's neighbors.

3. Let G′ = f(G) be the disjoint union of all these n subgraphs:

   G′ = $G_1 \cup G_2 \cup \cdots \cup$ G_n.

4. This construction is polynomial ($O(n^3)$ time).

Correctness:

- **(⇒)** If G has an (i + 1)-clique S, pick any vertex v_j ∈ S. The other i vertices of S form an i-clique in G_j. So G′ has an i-clique.
- **(⇐)** If G′ has an i-clique in some G_j, then {v_j} plus those i vertices form an (i+1)-clique in G.

Grading:

- [1 mark] Correct Karp reduction (disjoint union of neighborhood subgraphs) with reasoning. Partial credit (0.5) if Turing reduction confused with Karp.

2.4 Spot the Error and Suggest Modifications (2 Marks)

The Flawed "Proof":

"Consider induction on the clique size k. The base case is k = 1, and clearly CLIQUE_1 ∈ P. Also, you just proved that if CLIQUE_k ∈ P, then CLIQUE_k+1 ∈ P. From induction, we know ∀k : CLIQUE_k ∈ P. Thus P = NP."

1. The Error

The final conclusion "Thus P = NP" is a non-sequitur. The inductive argument correctly shows that each CLIQUE_k (for fixed k) is in P, but not that the general CLIQUE problem (where k is input) is in P.

Explanation:

- For fixed k, CLIQUE_k runs in O(n^k), which is polynomial in n.
- The general CLIQUE problem takes ⟨G, k⟩ as input; runtime O(n^k) is exponential in the input size (since k is variable, encoded in log k bits).
- Hence, proving $\forall k$: CLIQUE_k $\in$ P does not imply CLIQUE $\in$ P.

The author confuses a family of fixed-parameter problems (each in P) with the general parameterized version.

2. Correction

- The inductive argument is correct for fixed k.
- The step claiming P = NP must be removed—it does not follow.
- The proof cannot be modified to prove P = NP.

Grading:

- [2 marks] Correctly identifies the error as a false conclusion and explains the difference between k as a constant vs. an input variable.