Problem 1. Determine unique min-cut.

Solution:
Algorithm Description:
First compute a minimum $s-t$ cut (or min-cut) C, and let the cut capacity be $|C|$. Let $e_1, e_2, ..., e_k$ be the k edges in C. For each e_i, try increasing the capacity of e_i by 1 and compute a minimum cut in the new graph. Let the new minimum cut be C_i, and denote its capacity (in the new graph) as $|C_i|$. If any of the new cuts have a capacity equal to the original cut capacity $|C|$, a unique min-cut does not exist in the graph.

Proof of Correctness:
Consider a graph $G = (V, E)$ with a min-cut C, with a capacity $|C|$. Without loss of generality, we increase the capacity of an edge e_i by 1 and obtain a new graph G_i. We claim that there is a unique min-cut in the original graph G if for every i, the cut capacity $|C_i|$ in the new graph G_i is greater than $|C|$.

We first note that, for all i, $|C| \leq |C_i|$. The new cut capacity after increasing the edge capacity by 1 cannot be smaller than the cut capacity of the original graph.

For any i, if the cut capacity of the new graph G_i is equal to the original cut capacity, then we could obtain the original cut capacity by not including e_i in the cut. Therefore, we do not have a unique min-cut. Conversely, if there are multiple min-cuts for the graph, increasing the capacity of edge e_i will not affect the cut capacity as we can obtain the same cut capacity with a set of cut edges that do not include e_i.

Therefore, a graph has a unique min-cut, if $|C| < |C_i|$ for all i.

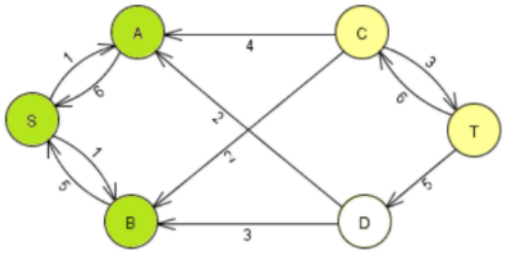
Complexity Analysis:
We can compute the min-cut of the original graph in $O((|E|)|C|)$ by using the Ford Fulkerson algorithm. After increasing the edge capacity by 1, we can compute the new min-cut in $O(|V| + |E|)$ time, as this involves finding an s-t path in the updated residual graph and updating the min-cut. We compute the new min-cut k times, therefore, the run time is $O((|V| + |E|)k)$ In total, the run time is $O(|E|(|C| + k))$

Problem 2. (DPV 7.17)

Solution:

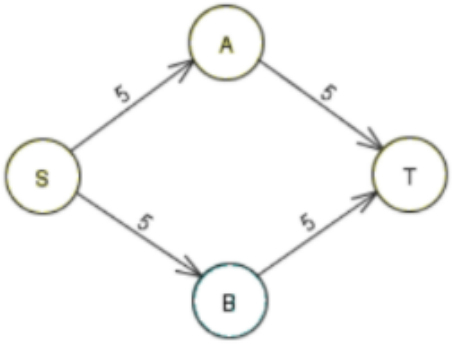
a. **Max flow and Min cut:**
The max flow value is 11. A min cut is $L = \{S, A, B\}, R = \{C, D, T\}$.

b. **Residual Graph:**
The residual graph is shown below. The vertices reachable from S are colored green and vertices from which T is reachable are colored yellow.



c. **Bottleneck Edges:**
The edges (A, C) and (B, C) are bottleneck edges.

d. **Example of network without bottleneck edges:**
The following graph has no bottleneck edges.



e. **Algorithm Description:**
Run the network flow algorithm and find the final residual network. For each edge (u, v) in the original graph, check in the residue network whether the vertex u of the edge is reachable from S and the vertex v of the edge can reach T. We can use a depth first search algorithm to check the reachability. If the paths from S to u and v to T exist, then edge (u, v) is a bottleneck edge. This is true because if we increase the capacity of the edge (u, v), then we get an edge (u, v) in the residual network. Using this edge, we can find an augmenting path from S to T resulting in an increase of flow value.

Proof of Correctness: We know that the existence of a path between vertices u and v in the residual graph indicates that flow can be augmented along that path. I.e., if vertex v is reachable from u, then the flow from u to v can be augmented. We use this property in this proof. Consider any edge (u, v) found by the algorithm such that u is reachable from s and t is reachable from v. Since the algorithm only checks for those edges that were present in the original graph, we know that edge (u, v) had some initial capacity that is now being fully used (as the edge does not exist in the residual graph). By the property stated earlier, there is some augmenting flow possible from s to u and v to t. Thus, increasing the capacity of edge (u, v) would introduce a new edge (u, v) in the residual graph with some non-zero capacity, making t reachable from s through edge (u, v). This new path from s to t is an augmenting flow, along which flow can be increased. Thus, (u, v) is a bottleneck edge. Consider the case when either u is not reachable from s or t is not reachable from v, but edge (u, v) is being used at maximum capacity. In this case, increasing the capacity of edge (u, v) will not increase flow from s to t, and hence (u, v) is not a bottleneck edge and is not found by the algorithm. Thus, the algorithm correctly finds only and all bottleneck edges in the graph.

Complexity Analysis: We can find all the vertices reachable from S in the residual graph in $O(|V| + |E|)$ time by a traversal algorithm like DFS. To find all the vertices that can reach T, we need only reverse the graph, which takes $(|V| + |E|)$ time, and then find all vertices reachable from T in the reversed graph, which also takes $O(|V| + |E|)$ time. Then checking each edge (u, v) to find those edges where u is in the set of vertices reachable from S and v is in the set of vertices that can reach T takes $O(|E|)$ time. Therefore, this algorithm takes $O(|V| + |E|)$, i.e., linear time, overall, in addition to the running time of the network flow algorithm for generating the final residual graph.

Problem 3: Flow Design Problem

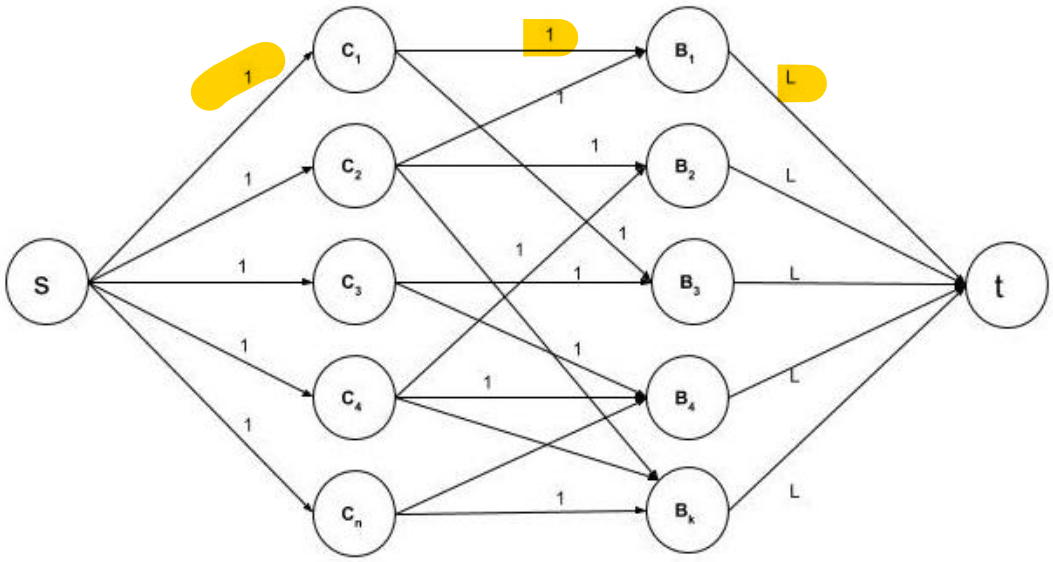
Solution:

Algorithm Description:

We build the following network: we first connect a client C_i to a base station B_j if the client is within a distance of R units from the base station. This directed edge (C_i, B_j) has a capacity of 1. We then connect a super-source node S to every client C_i with an edge of capacity 1. We connect every base station B_j to a super-sink node T with an edge of capacity L, the load parameter of each base station.

We can execute any network flow algorithm like Ford Fulkerson algorithm and obtain a maximum flow. If the flow saturates all outgoing edges from the source, i.e. if the maximum flow $F = n$, then we can connect every client to some base station.

Flow Diagram: The flow network is given below.



Proof of Correctness:

The following two claims will establish the proof of correctness.

Claim 1: If there is a way to connect all clients to the base stations, then there is a max-flow equal to n, the number of clients.

Proof: We prove this claim by constructing a flow for the graph $G = (V, E)$ with value n. We define the flow function as follows:

$$f(s, C_i) = 1 \text{ for } 1 \leq i \leq n$$
$$f(B_j, t) \leq L \text{ for } 1 \leq j \leq k$$
$$f(C_i, B_j) = 1 \text{ for all } (C_i, B_j) \text{ that are connected in the final solution}$$
$$f(C_i, B_j) = 0 \text{ for all other } (C_i, B_j) \text{ in } E$$

It is easy to check that the flow is a valid flow and its value is n. Since this flow saturates the outgoing edges from the source, the max-flow is equal to n.

Claim 2: If there is a max-flow equal to n, then the algorithm outputs a valid solution that connects each client to some base station, satisfying the constraints.

Proof: Let the max-flow be $F = n$. Since each outgoing edge has a capacity 1 and there are n edges, each edge is saturated with a flow of 1. By flow conservation property and unit edge capacities, only one of the outgoing edges from each client has a flow of 1 to some base station. Since the outgoing flow from a base station is limited to the capacity L, we cannot have a case where the number of clients connected to a base station is greater than L. Since the flows are conserved at all points, it must be the case that each edge (C_i, B_j) with a unit flow must be a part of the solution. We therefore note that the algorithm outputs some valid solution, and not necessarily a unique solution, since it is possible to have multiple valid solutions for the same constraints.

Complexity Analysis: The number of nodes in the network graph, $|V| = O(n + k)$. The number of edges in the network graph, $|E| = O(nk)$. One can apply Ford-Fulkerson algorithm to get $((|V|+|E|)n)$, which gives us an overall time bound of $O(n^2k)$.

Problem 4.(DPV 8.12) NP-Completeness

Solution:

a. **Proof:** To show that k-SPANNING TREE is a search problem, we need to show that it is possible to verify a solution in polynomial time. Given a spanning tree T for a graph G we need to verify that it is indeed a tree, that each edge in T is present in G and that all vertices of G are present in T and have degree k. All this can be done by a single DFS on T, comparing each edge with the corresponding edge in G, which takes polynomial time in total.

b. **Proof:** Consider the 2-SPANNING TREE problem. We are required to find a tree with each vertex having degree at most 2. However, such a tree must be a path, since creating a branch at any vertex makes the degree of that vertex as 3. Also, if the tree spans the graph it must be a an undirected Rudrata path. Hence, this problem is same as the undirected Rudrata path problem which is known to be NP-Complete. We now reduce the 2-SPANNING TREE problem to the k-SPANNING TREE problem for $k > 2$.
Given a graph $G = (V, E)$, at each vertex $v \in V$, we add $k-2$ buffer vertices $v_1, \cdots, v_{k-2}$ connected only to v. We add a fresh set of buffer vertices for each original vertex in the graph. Call this new graph G_0. It is easy to see that a k-spanning tree of G_0 must contain all the buffer vertices as leaves, since they all have degree 1. Removing, these gives a 2-spanning tree of G. Similarly, adding $k-2$ buffer leaves to each vertex in any 2-spanning tree of G, will give a k-spanning tree of G_0. Hence, the k-SPANNING TREE problem is NP-Complete for every $k \geq 2$.

Problem 5. (DPV 9.3) Approximation Algorithm.

Solution:
Proof: Let T be the minimum Steiner tree having cost C. Then we can follow the shape of the Steiner tree to obtain a tour of cost $2C$ which passes through all the vertices in the Steiner tree. Let i, j, k be adjacent vertices in the path. Using the triangle inequality, we know that $d_{ik} \leq d_{ij} + d_{jk}$. Hence, we can bypass an intermediate vertex j and connect i and k directly if we want, without increasing the cost. Using this trick, we can bypass all the vertices not in V' that are present in the path, and also the vertices of V' which are being visited twice. This gives a path of cost at most $2C$, which passes through all the vertices of V' exactly once. Hence, this path is a spanning tree for V' of cost $2C$. This implies $cost(MST) \leq 2C$, thus giving the desired approximation guarantee.

Problem 6. (DPV 7.3) Linear Programming

Solution:
Variables:
Let x_1, x_2 and x_3 represent the amount of material 1, 2 and 3, in cubic meters, loaded on the ship, respectively.

Constraints:
We have the following constraints on these amounts.
$0 \leq x_1 \leq 40$
$0 \leq x_2 \leq 30$
$0 \leq x_3 \leq 20$

Since the ship is only able to carry at most 60 cubic meters of material, we must have
$x_1 + x_2 + x_3 \leq 60$
Also, the maximum weight limit on the cargo is 100 tons, which gives the following constraint.
$2x_1 + x_2 + 3x_3 \leq 100$

Objective Function:
We want to maximize the revenue:
Maximize $z = 1000x_1 + 1200x_2 + 12000x_3$

Problem 7. Backtracking.

Solution:

a. **Final Coloring:**

- **Strategy A**:

$$L, O, L, T, O, T$$

- **Strategy B**:

$$L, T, O, L, T, O$$

b. **Total number of nodes explored:**

- **Strategy A**:

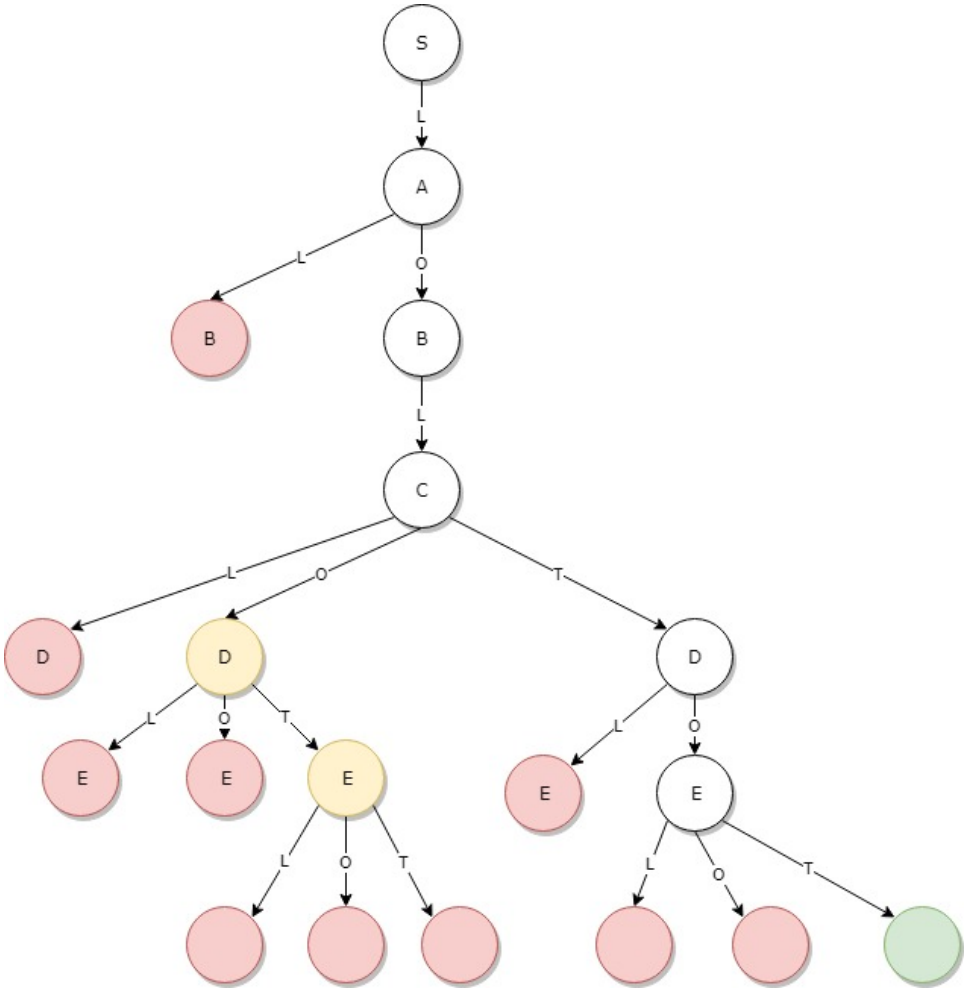
18

- **Strategy B**:

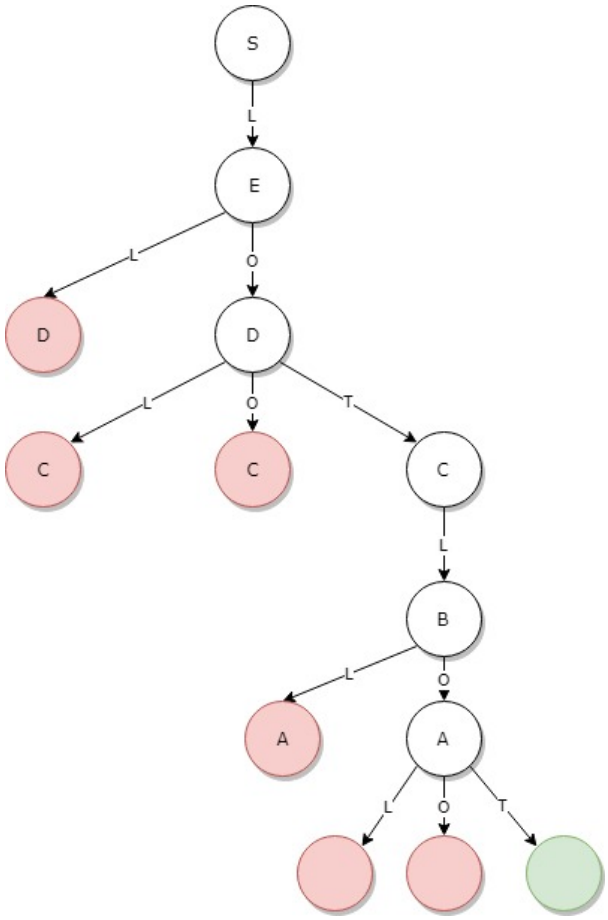
12

The tree below illustrates the execution of the back tracking algorithm. On each node an edge coming out suggests that the color was applied to the node it is coming out from. Nodes with red indicate that an invalid coloring had been done. and hence the algorithm back tracked. A node in orange indicates that there were no other possible actions from that node.

Strategy A



Strategy B



The total times number of times a color was tried out on the algorithm is the number of those edges. And the valid order is the valid path found.