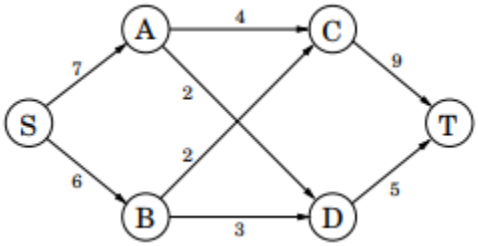
Q7.17

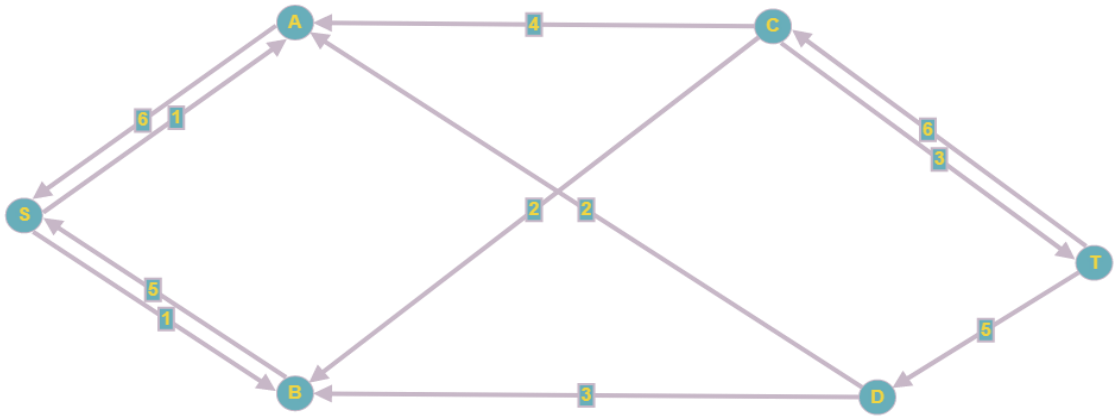
Consider the following network (the numbers are edge capacities).



(a) Find the maximum flow *f* and a minimum cut.

The maximum flow in the above graph is 11. The minimum cut is the cut made straight down the center of the graph such that the two halves are {S, A, B} and {C, D, T}.

(b) Draw the residual graph G_f (along with its edge capacities). In this residual network, mark the vertices reachable from S and the vertices from which T is reachable.

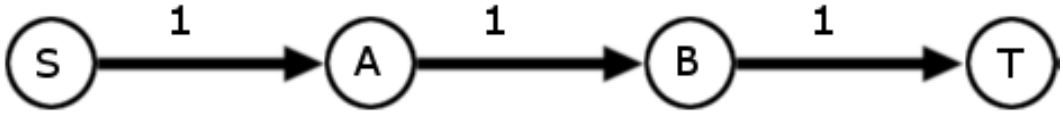


The vertices reachable from S are {S, A, B}. Vertices C and T can reach T.

(c) An edge of a network is called a bottleneck edge if increasing its capacity results in an increase in the maximum flow. List all bottleneck edges in the above network.

The edges (A, C) and (B, C) are bottleneck edges.

(d) Give a very simple example (containing at most four nodes) of a network which has no bottleneck edges.



(e) Give an efficient algorithm to identify all bottleneck edges in a network. (Hint: Start by running the usual network flow algorithm, and then examine the residual graph.)

1. Run the Edmonds–Karp variation of the Ford-Fulkerson algorithm, making sure that on each iteration, a BFS is used to pick the s-t path with the fewest edges. Find the residual graph. This variation of the algorithm runs in $O(V * E^2)$ time.
2. Use a search algorithm to find the vertices reachable from s in the residual graph – linear time.
3. Reverse the residual graph and use another search algorithm on t to find the vertices that can reach t – linear time.
4. By the definition of a residual graph, we know that the presence of a path from one of the vertices found in step 2 to one of the vertices found in step 3 means that flow can be augmented along that path, thus forming a path from s to t in the residual graph. Thus, the connections between vertices in the "step 2" group and vertices in the "step 3" are all bottleneck edges. Checking this property takes linear time.

   The final runtime for the algorithm is $O(V * E^2)$

Q7.21

An edge of a flow network is called critical if decreasing the capacity of this edge results in a decrease in the maximum flow. Give an efficient algorithm that finds a critical edge in a network.

1. Run the Edmonds–Karp variation of the Ford-Fulkerson algorithm, making sure that on each iteration, a BFS is used to pick the s-t path with the fewest edges. Find the residual graph. This variation of the algorithm runs in $O(V * E^2)$ time.
2. We need only consider edges at max capacity as potential candidates for critical edges. Identifying these edges takes linear time.
3. Of the edges at full capacity in the residual graph, use a DFS or another search algorithm to see if there exists another path between the vertices that make up the edge. If so, then flow can potentially be rerouted and that edge is therefore not critical. If there is not a path, then flow cannot be rerouted and the path is critical. A DFS must be run on potentially V vertices, which results in a runtime of $O(V(V + E))$.

The final runtime of the algorithm is $O(V*E^2 + V^2 + V*E) \rightarrow O(V*E^2)$.

Q8.4

Consider the CLIQUE problem restricted to graphs in which every vertex has degree at most 3. Call this problem CLIQUE-3.

(a) Prove that CLIQUE-3 is in NP.

Given a set of vertices believed to be in a clique, we can check to see that every vertex is connected to one another in N^2 time -- if there are N vertices, then for every vertex we just need to check for an edge to the other N-1 vertices.

(b) What is wrong with the following proof of NP-completeness for CLIQUE-3?
We know that the CLIQUE problem in general graphs is NP-complete, so it is enough to present a reduction from CLIQUE-3 to CLIQUE. Given a graph G with vertices of degree ≤ 3, and a parameter g, the reduction leaves the graph and the parameter unchanged: clearly the output of the reduction is a possible input for the CLIQUE problem. Furthermore, the answer to both problems is identical. This proves the correctness of the reduction and, therefore, the NP-completeness of CLIQUE-3.

This reduction moves in the wrong direction. To show that Clique-3 is NP-complete, we need to find a reduction from clique to clique 3. The above proof attempts to reduce clique 3 to clique.

(c) It is true that the VERTEX COVER problem remains NP-complete even when restricted to graphs in which every vertex has degree at most 3. Call this problem VC-3. What is wrong with the following proof of NP-completeness for CLIQUE-3? We present a reduction from VC-3 to CLIQUE-3. Given a graph G = (V, E) with node degrees bounded by 3, and a parameter b, we create an instance of CLIQUE-3 by leaving the graph unchanged and switching the parameter to |V | − b. Now, a subset C ⊆ V is a vertex cover in G if and only if the complementary set V − C is a clique in G. Therefore G has a vertex cover of size ≤ b if and only if it has a clique of size ≥ |V| − b. This proves the correctness of the reduction and, consequently, the NP-completeness of CLIQUE-3.

The line about "a subset C ⊆ V is a vertex cover in G if and only if the complementary set V − C is a clique in G" is false. A subset C in V is a vertex cover in G if the complementary set V - C is an independent set in G.

(d) Describe an $O(|V|^4)$ algorithm for CLIQUE-3.

We know that because the maximum degree is 3, the max size of a clique can be 4. For every potential clique, the max number of edges can be 6. We can just choose every combination of 4 elements from the number of vertices (so v * v-1 * v-2 * v-3) combinations, do a constant amount of work checking for potential edges – so O(v^4). We can then repeat the process for smaller potential cliques, but the max size of 4 dominates the runtime. Thus the final runtime is O(v^4).

Q8.15

Show that the following problem is NP-complete.
MAXIMUM COMMON SUBGRAPH
Input: Two graphs G1 = (V1, E1) and G2 = (V2, E2); a budget b.
Output: Two set of nodes $V'_1 \subseteq V_1$ and $V'_2 \subseteq V_2$ whose deletion leaves at least b nodes in each graph, and makes the two graphs identical.

Let's consider the subgraph isomorphism problem which we know to be NP hard. This problem takes two graphs and attempts to see if one graph contains a subgraph that is isomorphic to the other. Two graphs are isomorphic if their vertices and edges are same. Let G1 and G2 be the inputs to subgraph-isomorphism. Then we provide (G1, G2, |V1|) as input to the maximum common subgraph. G2 has a subgraph that is isomorphic to G1 if can have subsets W1 in V1 and W2 in V2 whose deletion leaves behind at least a number of vertices equal to at least V1 in each graph and makes the two graphs isomorphic to one another. This is a polynomial time reduction, so if there were a polynomial time algorithm for the maximum common subgraph problem, then we would have one for the Subgraph-Isomorphism problem.