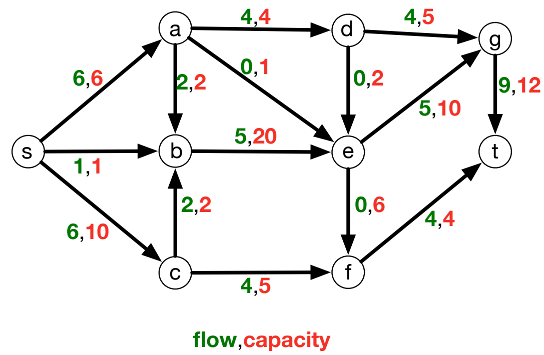
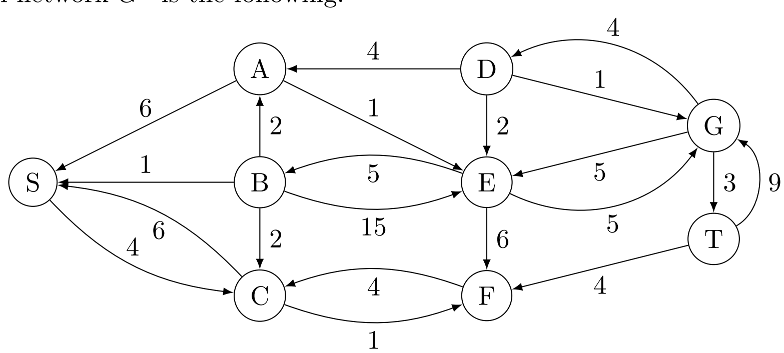
Practice problems:

1. [DPV] Problem 7.10 (max-flow = min-cut example)

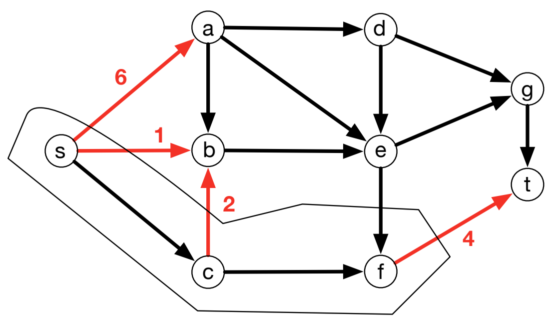
Here is a max flow in the given flow network:



The residual network G^f is the following:



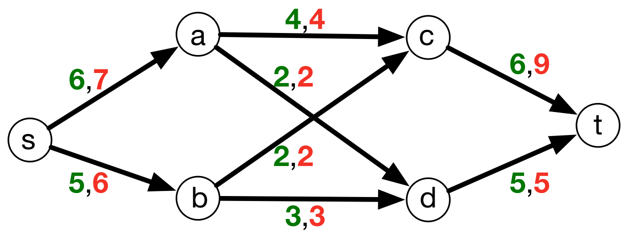
Looking at the residual network G^f, the set L of vertices reachable from s in G^f is $L = \{s, c, f\}$. This set L has capacity $13 = 6 + 1 + 2 + 4$. Note the capacity of the cut is determined by the original capacities, it does not depend on the flow. The capacity of this st-cut matches the size of the flow f and hence f is a max-flow and L defines a min-st-cut. Here is an illustration of this min-st-cut:



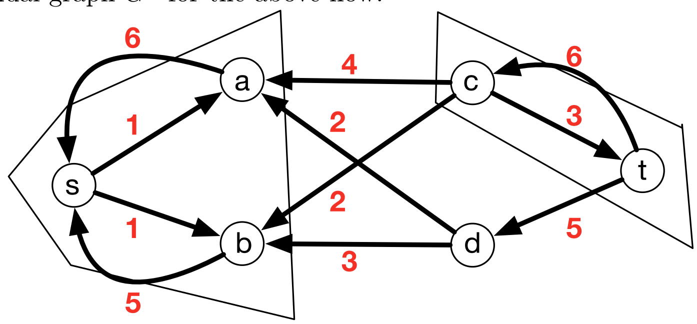
2. [DPV] Problem 7.17 (bottleneck edges)

Parts (a) and (b):

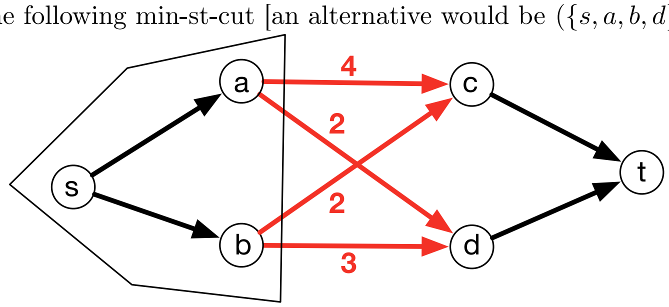
Here is the max flow in the given flow network:



Here is the residual graph G^f for the above flow:



In G^f the set of vertices reachable from s is $\{s, a, b\}$ and the set of vertices that can reach t is $\{c, t\}$. This gives the following min-st-cut [an alternative would be $(\{s, a, b, d\}, \{c, t\})$]:



Notice that the set $\{s, a, b\}$ has capacity $11 = 4 + 2 + 2 + 3$ which matches the size of the above flow.

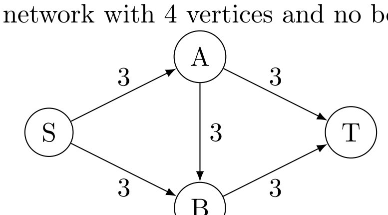
Part (c):

An edge $\overrightarrow{uv}$ in the original flow network G is a *bottleneck edge* if increasing its capacity results in an increase in the size of the maximum flow.

There are two bottleneck edges in the above network, they are the edges $\overrightarrow{ac}$ and $\overrightarrow{bc}$.

Part (d):

Here is an example of a flow network with 4 vertices and no bottleneck edges:



Alternatively, in the flow network from question 7.17, if the capacity of the edge $\overrightarrow{ct}$ was reduced from 9 to 6 then there will be no bottleneck edges in this flow network.

Part (e):

Here is the general algorithm for finding all of the bottleneck edges in the flow network G.

We start by finding a maximum flow f for the flow network G. Consider an edge $\overrightarrow{vw}$ in the flow network G. Increasing the capacity of $\overrightarrow{vw}$ results in an increase in maximum flow value if and only if there exists a path from s to v and a path from w to t in G^f. This is because if there exists these two paths then more flow can be sent from s to u, then along the edge $\overrightarrow{vw}$, and finally from v to t.

Therefore, our algorithm for finding bottleneck edges is as follows:

1. Find a maximum flow f on G.
2. Run Explore from s in G^f. Let S be the set of vertices reachable from s in G^f.
3. Run Explore from t in the reverse graph of G^f. Let T be the set of vertices reachable from t in the reverse graph of G^f; note the set T are those vertices which can reach t in G^f.
4. For each $\overrightarrow{vw} \in E(G)$, output $\overrightarrow{vw}$ as a bottleneck edge if $v \in S$ and $w \in T$.

Since steps 2, 3, and 4 take $O(|V| + |E|)$ time, the running time is dominated by the running time of the maximum flow algorithm in step 1.

Note that this algorithm looks for a path $s \to v$ and $w \to t$. What if these two paths share one or more edges? Then, the joined path will have one or more cycles. So, we can drop that cycle (or cycles) and get a shorter path from $s \to t$, but will this path still go through (v, w)? If one of the cycles contains edge $e = (v, w)$, then we have an augmenting path in G^f not using e, which would mean f is not a max flow. Hence, e cannot be in any of the cycles, so our algorithm works.

3. [DPV] Problem 7.19 (verifying max-flow)

Given a flow network G and a flow f, note that f is a maximum flow iff there is no augmenting path from s to t in the residual graph. Hence to verify that f is of maximum size we first construct the residual graph G^f. We then run Explore from s on G^f to check if there is a path from s to t. If t is reachable from s then there is an augmenting path and hence f is not of maximum size. On the other hand if t is not reachable from s in G^f then we know that f is a max-flow.

4. Bipartite perfect matching

For a bipartite graph $G = (V_1 \cup V_2, E)$ where $|V_1| = |V_2| = n$ a *perfect matching* is a subset S of edges where each vertex is incident exactly 1 edge in S. In other words, it's a matching of size n. Given a bipartite graph G show how to determine if G has a perfect matching by a reduction to the max-flow problem. In other words, given G define an input flow network to the max-flow problem. Then given a max-flow for this input how do you determine if the original graph G has a perfect matching or not? What is the running time of your algorithm?
(For hints see [DPV] Chapter 7.3 (Bipartite matching) and the beginning of Problem 7.24.)

Given the bipartite graph G we define the following flow network $\overrightarrow{G}$:

- Add a source node s and add an edge from s to each vertex in V_1. Each of these edges is given capacity 1.
- For each edge $(v, w) \in E$ in the original bipartite graph G where $v \in V_1$ and $w \in V_2$ we direct the edge from v to w and assign it capacity 1. Thus all edges of G now point from V_1 to V_2.
- Add a sink node t and add an edge from each vertex in V_2 to t. Each of these edges is given capacity 1.

Given a max-flow on this flow network, each of the directed edges from V_1 to V_2 that have flow along them are put in the matching M. Note that each vertex $v \in V_1$ has at most one edge incident to it in M since v has one incoming edge and this incoming edge has capacity 1 so at most one unit of flow comes out of v. Similarly each $w \in V_2$ has at most one edge incident to it in M since w has one outgoing edge and it is of capacity 1.

If the size of the flow is $= n$ then $|M| = n$ and M is a perfect matching.

We run a max-flow algorithm. We can use the Edmonds-Karp algorithm which takes time $O(nm^2)$. Alternatively since all the capacities are integers we can use the Ford-Fulkerson algorithm which takes time $O(mC)$ where C is the size of the max-flow. Notice that $C \leq n$ and hence the Ford-Fulkerson algorithm takes time $O(nm)$ which is faster than Edmonds-Karp in this case.