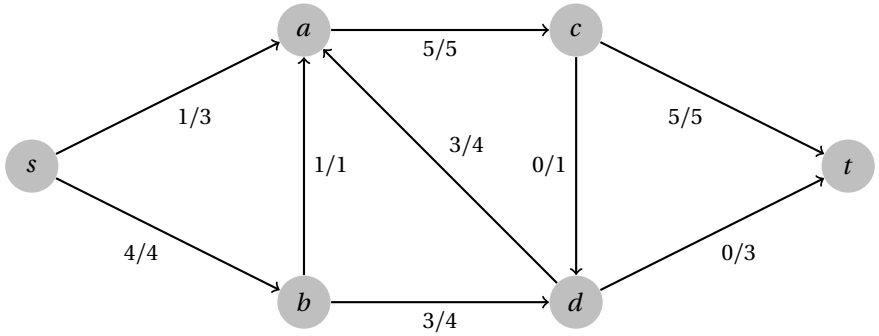
Week 9 Studio Sheet

(Solutions)

Useful advice: The following solutions pertain to the theoretical problems given in the studio classes. You are strongly advised to attempt the problems thoroughly before looking at these solutions. Simply reading the solutions without thinking about the problems will rob you of the practice required to be able to solve complicated problems on your own. You will perform poorly on the exam if you simply attempt to memorise solutions to the studio problems. Thinking about a problem, even if you do not solve it will greatly increase your understanding of the underlying concepts. Solutions are typically not provided for Python implementation questions. In some cases, pseudocode may be provided where it illustrates a particular useful concept.

Studio Problems

Problem 1. Consider the following flow network. Edge labels of the form f/c denote the current flow f and the total capacity c.



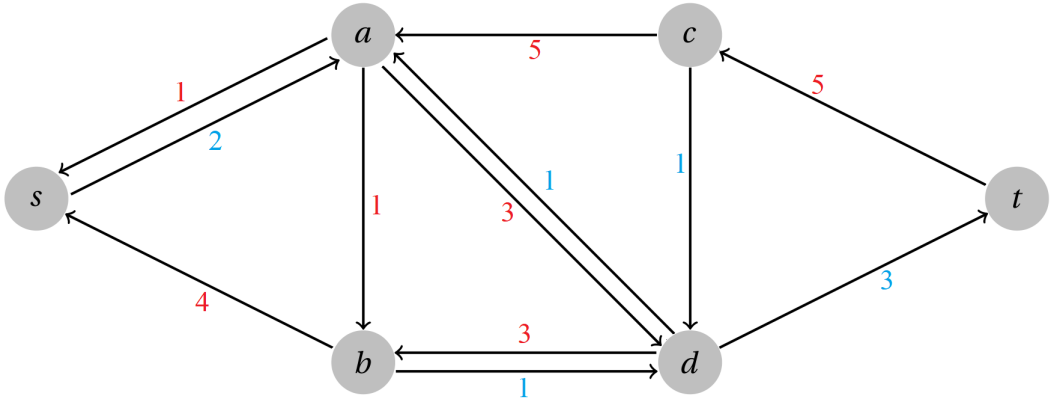
(a) Draw the corresponding residual network.

(b) Identify an augmenting path in the residual network and state its capacity.

(c) Augment the flow of the network along the augmenting path, showing the resulting flow network.

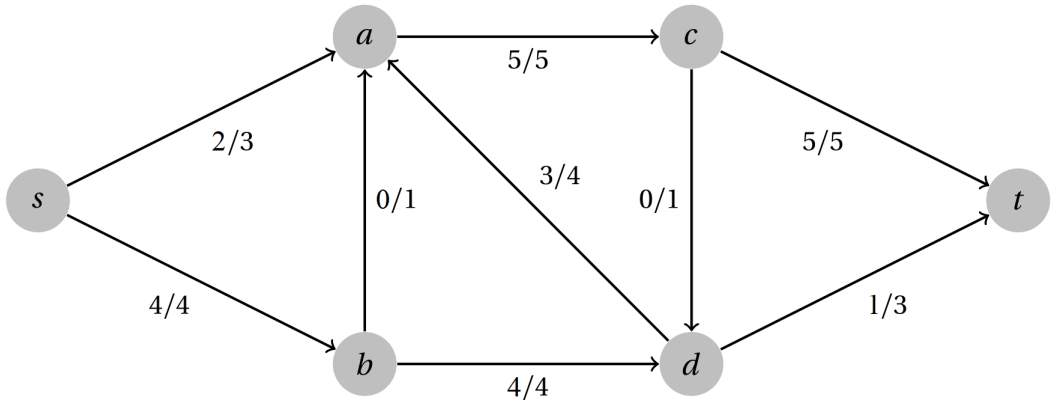
Solution

(a) Residual capacity is shown in blue, reversible flow is shown in red.



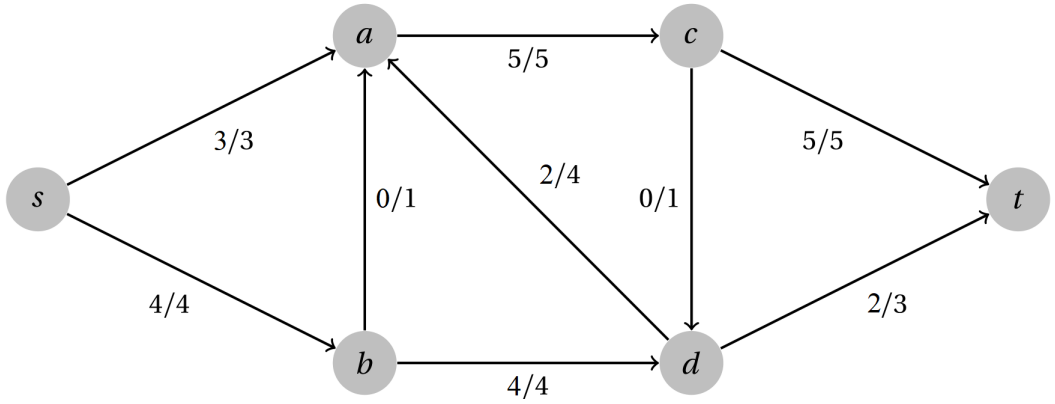
(b) One augmenting path is $s \to a \to b \to d \to t$ with capacity 1.

(c) Note that there are other possible augmenting paths and therefore other solutions.



Problem 2. Complete the Ford-Fulkerson method for the network in Problem 1, showing the final flow network with a maximum flow.

Solution



Problem 3. Using your solution to Problem 2, list the vertices in the two components of a minimum $s-t$ cut in the network in Problem 1. Identify the edges that cross the cut and verify that their capacity adds up to the value of the maximum flow.

Solution

To find the minimum cut after running the Ford-Fulkerson algorithm, we simply identify all vertices reachable from the source in the residual graph. These vertices will be on one side of the cut, and all remaining vertices will be on the other. In this graph, s has no outgoing edges in the residual graph, since the two edges outgoing from s are full. So the source is on one side of the cut, and everything else is on the other.
Component 1: $\{s\}$
Component 2: $\{a, b, c, d, t\}$
The flow across this cut is indeed 7, as expected.

Problem 4. Let G be a flow network and let f be a valid flow on G. Prove that the net outflow out of s is equal

to the net inflow into t.

Solution

Suppose the set of all vertices is V. We know that the flow of every cut is equal to the flow of the network. Therefore the capacity of the cut $(\{s\}, \{V - \{s\}\})$ - which is the flow out of s - must equal the capacity of the cut $(\{V - \{t\}\}, \{t\})$ - which is the capacity into t.

Problem 5. Consider a variant of the maximum network flow problem in which we allow for multiple source vertices and multiple sink vertices. We retain all of the capacity and flow conservation constraints of the original maximum flow problem. As in the original problem, all of the sources and sinks are excluded from the flow conservation constraint. Describe a simple method for solving this problem.

Solution

We create a new graph by adding a "super-source" and "super-sink" vertex. The super source is connected to each source by an edge with capacity equal to the total capacities of all the outgoing edges from that source. Similarly, each sink is connected by an edge to the super sink, and the capacity of that edge is equal to the total capacities of all incoming edges to that sink. We then solve the flow problem on this new graph as normal.

Problem 6. Given a list of n integers $r_1, r_2, ..., r_n$ and m integers $c_1, c_2, ..., c_m$, we wish to determine whether there exists an $n \times m$ matrix consisting of zeros and ones whose row sums are $r_1, r_2, ..., r_n$ respectively and whose column sums are $c_1, c_2, ..., c_m$ respectively. It is assumed that the sum of the r's is equal to the sum of the c's (otherwise it is trivially impossible to satisfy the constraints). Describe an algorithm for solving this problem by using a flow network.

Solution

Construct a flow network as follows: Create two sets of vertices, A and B. Each vertex in A $(a_1, a_2, ...a_n)$ corresponds to one of the n rows, and each vertex in B $(b_1, b_2, ...b_m)$ corresponds to one of the m columns. Add a source and a sink. For $1 \leq i \leq n$, add an edge from the source to a_i with capacity r_i. For each $1 \leq j \leq m$, add an edge from vertex b_j to the sink with capacity c_j. Add edges $e_{i,j}$ between every pair of vertices (a_i, b_j) with capacity 1. Determine the maximum flow in this network. If all the edges from the source are full, then we can construct such a matrix, and moreover, the elements $A_{i,j}$ of the matrix are equal to the flows along $e_{i,j}$. If any edge from the source is not full, such a matrix does not exist.

To see why this method works, consider the flows into and out of the a_i vertices. We notice that a row has r_i 1s iff the vertex a_i has a total outflow of r_i (because each edge out of a_i corresponds to one of the values in column i, and all the edges from a_i have capacity 1), but this is only possible if the flow from the source to a_i is exactly r_i because the only edge that flows into a_i is from the source. These edges have capacity r_i, so they all need to be full.

Problem 7. Consider the problem of allocating a set of jobs to one of two supercomputers. Each job must be allocated to exactly one of the two computers. The two computers are configured slightly differently, so for each job, you know how much it will cost on each of the computers. There is an additional constraint. Some of the jobs are related, and it would be preferable, but not required, to run them on the same computer. For each pair of related jobs, you know how much more it will cost if they are run on separate computers. Give an efficient algorithm for determining the optimal way to allocate the jobs to the computers, where your goal is to minimise the total cost.

Solution

Create a graph with (bidirectional) edges between related jobs, whose capacities are the cost to separate them. Add edges from s to all jobs with capacity *cost to run on computer 1*, and edges from all jobs to t with capacity *cost to run on computer 2*. The minimum cut will be the best total cost since you are paying to either put the job on computer 1 or 2, and paying for related jobs that are cut from each other when assigned to different computers.

Problem 8. Consider a variant of the maximum network flow problem in which vertices also have capacities. That is for each vertex except s and t, there is a maximum amount of flow that can enter and leave it. Describe a simple transformation that can be made to such a flow network so that this problem can be solved using an ordinary maximum flow algorithm[1].

Solution

For each non-source non-sink vertex v, do the following: Create two new vertices v_{in} and v_{out}. Take all the inflowing edges to v and replace v with v_{in}. Take all the outflowing edges from v and replace v with v_{out}. Create an edge from v_{in} to v_{out} with capacity equal to the capacity of v. Delete v from the network.

Now we can solve for the maximum flow in this graph, and the solution will be the maximum flow in the original graph as well.

Problem 9. Two paths in a graph are *edge disjoint* if they have no edges in common. Given a directed graph, we would like to determine the maximum number of edge-disjoint paths from vertex s to vertex t.

(a) Describe how to determine the maximum number of edge-disjoint $s-t$ paths.

(b) What is the time complexity of this approach?

(c) How could we modify this approach to find *vertex-disjoint paths* (i.e. paths with no vertices in common)?

Solution

(a) Given a directed graph G, create a flow network G' from G as follows: Let s be the source, and t be the sink. Give every edge a capacity of 1. Now we can run the maximum flow algorithm, and the resulting flow is the number of *edge-disjoint* paths.

This works because the number of *edge-disjoint* paths that pass through a vertex v is at most $\min(\text{indegree}(v), \text{outdegree}(v))$. The maximum flow through a vertex v is the same value, because all edges are capacity 1.

(b) The time complexity of Ford-Fulkerson is bounded by $O(Ef)$ where E is the number of edges and f is the maximum flow. Here, the maximum flow is the outdegree of s, so the complexity is $O(E \times \text{outdegree}(s))$ which is bounded by $O(EV)$.

(c) Create the flow network as specified in part a), but add capacities for each vertex, and then perform the transform described in the solution to problem 8. This ensures that the flow through each vertex is at most 1, whereas before the flow through a vertex was equal to the number of paths through it.

Supplementary Problems

Problem 10. Implement the Ford-Fulkerson method using:

[1] Do not try to modify the Ford-Fulkerson algorithm. In general, it is always safer when solving a problem to reduce the problem to another known problem by transforming the input, rather than modifying the algorithm for the related problem.

- DFS for finding augmenting paths.
- BFS for finding augmenting paths.

Problem 11. You are a radio station host and are in charge of scheduling the songs for the coming week. Every song can be classified as being from a particular era, and a particular genre. Your boss has given you a strict set of requirements that the songs must conform to. Among the songs chosen, for each of the n eras $1 \leq i \leq n$, you must play exactly n_i songs of that era, and for each of the m genres $1 \leq j \leq m$, you must play exactly m_j songs from that genre. Of course, $\sum n_i = \sum m_j$. Devise an algorithm that given a list of all of the songs you could choose from, their era and their genre, determines a subset of them that satisfies the requirements, or determines that it is not possible.

Solution

Make a bipartite graph with vertices $u_1 \ldots u_i$ corresponding to eras, and $v_1 \ldots v_j$ corresponding to genres. Each song is represented by an edge from its genre to its era (of capacity 1 if you can only play each song once. How would you allow for each song to be played some other number of times at most?). Add a source with outgoing edges to each genre vertex u_k with capacity n_k (where $1 \leq k \leq n$). Add a sink vertex, with an edge from each vertex v_l to the sink with capacity m_l (where $1 \leq l \leq n$). Run Ford-Fulkerson on this graph, and if a flow of capacity $\sum n_i = \sum m_j$ is achieved, then it is possible to play songs in the required combination. Otherwise it is not. The flow along song edges tells you how many times to play each song.