Math 482: Linear Programming[1] Mikhail Lavrov

Lecture 27: Variations on Maximum Flow Problems

April 8, 2020 *University of Illinois at Urbana-Champaign*

1 Multiple sources and sinks

Our maximum flow problems have been solving the problem of transporting flow from a single source to a single sink. This is not necessarily how things work in applications: often, the network has multiple source nodes and multiple sink nodes.

Rather than develop a separate method of solving such problems (and other variations on the maximum flow problem), what we do is reduce them to the existing problem.

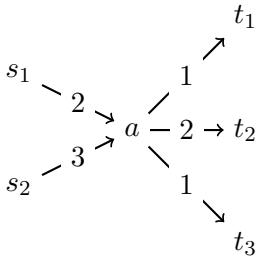
Suppose we are given a network with sources $s_1, s_2, \ldots, s_k$ and sinks $t_1, t_2, \ldots, t_\ell$. Otherwise, things are the same as before: there are arcs between the nodes, capacities on the arcs, and we'd like to maximize the flow from the sources to the sinks. In this problem, we don't care about which source directs flow to which sink: we're only transporting one kind of "stuff". Also, it doesn't matter which sources we use as a suppply, or which sinks receive it.

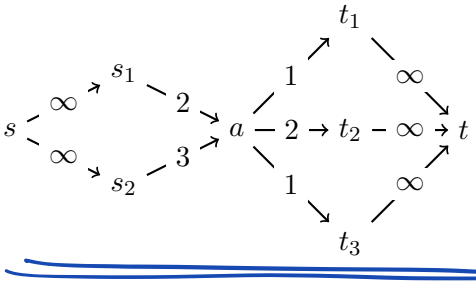
To solve this kind of problem, we simply add a new "super-source" s and a new "super-sink" t. We create arcs (s, s_i) from the super-source to every source, and arcs (t_i, t) from every sink to the super-sink. These arcs have infinite capacity: there is no limit to the amount of flow on them.

(If putting a capacity of ∞ on an arc bothers you, we could give these arcs some very large capacity: for example, the sum of all the capacities of the previously-existing arcs. The idea here is to just make the capacity large enough that it will never actually prevent us from sending more flow along one of these arcs.)

Now we can solve the ordinary s-to-t maximum flow problem in the new network, and get a solution to the multiple-sink to multiple-source maximum flow problem in the original network without s and t. Of course, we should ignore s and t when we go back to the old problem.

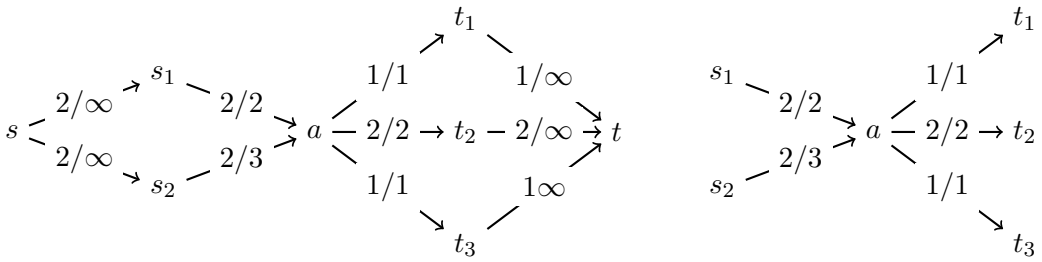
This is in general how the reductions we'll study today go. Starting from some new, weird kind of problem (left), we construct a familiar kind of problem (right):





[1] This document comes from the Math 482 course webpage: https://faculty.math.illinois.edu/~mlavrov/courses/482-spring-2020.html

Then, once we solve the new problem (left), we do some transformation to return to a solution to the old problem (right). In this case, the transformation is just "forget about s and t", but it can be more general than that.



2 Nodes with supply and demand

There's a different kind of setup for problems with multiple sources and sinks. Rather than having these multiple locations where an "unlimited supply of stuff" gets created, and similarly multiple locations where it gets destroyed, we can imagine nodes with a supply and a nodes with a demand. If a node has a supply of 3 units, it can send out 3 more flow than it receives; if a node has a demand of 3 units, it wants to receive 3 more flow than it sends out.

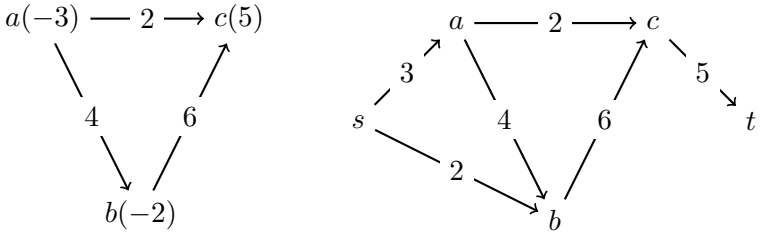
To formalize this notion, let's define the *excess* at a node to be the difference between the amount of flow entering it and the amount of node leaving. The excess of a node k with respect to a feasible flow $\mathbf{x}$ is

$$\Delta_k(\mathbf{x}) := \sum_{i:(i,k)\in A} x_{ik} - \sum_{j:(k,j)\in A} x_{kj}.$$

Usually, we put a flow conservation constraint on our network, which is asking that for all nodes $k \neq s, t$, we have $\Delta_k(\mathbf{x}) = 0$.

In this problem, we instead give each node k a value d_k representing the demand at that node. If d_k is negative, it instead represents a supply. We replace the flow conservation constraints by the requirement that for every node k, $\Delta_k(\mathbf{x}) = d_k$. There is no source or sink in this setup. Note that this only works if $\sum_{k\in N} d_k = 0$: if there is more demand in the network than there is supply, we can't make things work.

To model this as a classical maximum flow problem, we once again add a source s and a sink t to the problem. Whenever a node k has $d_k > 0$, we add an arc (k, t) with capacity d_k; whenever a node k has $d_k < 0$, we add an arc (s, k) with capacity $-d_k$. Like so:



The supply-and-demand problem is not an optimization problem: it's a feasibility problem. To see if there is a way to satisfy all the demands, we want to saturate all the arcs into t. (That is, reach a flow equal to the capacity along all these arcs.) So we want to find a maximum flow, and see if it has a sufficiently high value.

If we succeed, then we can go back to a solution to the original problem just by forgetting about nodes s and t, and the arcs in/out of those nodes. When the flow along those arcs is lost, the flow conservation constraint in the maximum flow problem gives us the desired excess at every node.

The same setup lets us solve a slightly more general variant of this problem, where the total supply *exceeds* the total demand, but we don't have to use all the supply. (In this case, saturating all the demand arcs simply won't saturate all the supply arcs.)

3 Feasible circulations

example in HW 8.4.

Another feasibility problem is the problem of finding a feasible flow in a network with no source or sink. Here, all the nodes have to satisfy flow conservation. A "circulation" is the word for a feasible flow in such a network, because the flow is not getting created or destroyed anywhere: it's just circulating.

You may ask, "But Misha, can't we just set $\mathbf{x} = \mathbf{0}$ and call it a day?" That's why feasible circulation problems have a special kind of constraint. Instead of asking that every x_{ij} is between 0 and an upper bound c_{ij}, we specify an interval $[a_{ij}, b_{ij}]$ for x_{ij} to be contained in. Usually, $0 \notin [a_{ij}, b_{ij}]$, and so the zero flow violates this constraint.

Here, we're going to go in two steps, and actually reduce this problem to the problem in the previous section (where nodes have supply and demand). The idea is this: if the flow x_{ij} has to lie in the interval $[a_{ij}, b_{ij}]$, then we reformulate this as "teleporting" a_{ij} flow out of node i, teleporting a_{ij} flow into node j, and then sending somewhere between 0 and $b_{ij} - a_{ij}$ flow along arc (i, j) as normal. This creates an artificial demand of a_{ij} at i and an artificila supply of a_{ij} at j.

Doing this to all arcs results in a supply-and-demand problem in which the demand at node k is given by the sum

$$d_k = \sum_{j:(k,j)\in A} a_{kj} - \sum_{i:(i,k)\in A} a_{ik}.$$

Solving the supply-and-demand problem produces a vector $\mathbf{x}$ where $0 \le x_{ij} \le b_{ij} - a_{ij}$ for all arcs $(i, j) \in A$. To return to the original problem, we replace x_{ij} by $x_{ij} + a_{ij}$.

This increases the flow out of each node k by $\sum_{i:(i,k)\in A} a_{ik}$, but also increases the flow into each node by $\sum_{j:(k,j)\in A} a_{kj}$, turning the demand constraint into flow conservation:

$$\sum_{i:(i,k)\in A} x_{ik} - \sum_{j:(k,j)\in A} x_{kj} = d_k \implies \sum_{i:(i,k)\in A} (x_{ik} + a_{ik}) - \sum_{j:(k,j)\in A} (x_{kj} + a_{kj}) = 0.$$

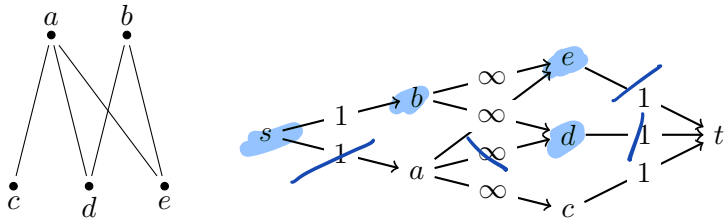
4 Bipartite matching, again

At the beginning of the graph theory unit, we talked about the bipartite matching problem. Though we wrote a linear program to solve it, it is much better to handle bipartite matching as a maximum flow problem.

Given a bipartite graph with vertices X on one side and Y on the other, here is how we turn the problem into a maximum flow problem.

1. For each edge (i, j) with $i \in X$ and $j \in Y$, add an arc (i, j) in the network with infinite capacity.[2]
2. Add a source s and a sink t. For each $i \in X$, add an edge (s, i) with capacity 1; for each $j \in Y$, add an edge (j, t) with capacity 1.

For example:



Here, the flow along an "inside" edge in the network is 1 if the corresponding edge is used in the matching. It seems like in theory some valid network flows will give fractional solutions, but neither the simplex method nor the Ford–Fulkerson algorithm will do something like that.

What's more interesting is that the dual problems also match up: a cut in the network gives us a vertex cover in the bipartite graph. More precisely, this is true of "reasonably good" cuts: cuts with finite capacity.

For a cut with finite capacity, whenever we include a vertex in X, we should also include all its neighbors in Y, to prevent an infinite edge from crossing the cut. So, for example, one possible cut (with capacity 3) is the cut $\{s, b, d, e\}$.

The edges crossing this cut are the edges $s \to a$, $e \to t$, and $d \to t$. This tells us the vertices we should include in the vertex cover: vertices a, d, and e. In general, for a cut (S, T), we include $X \cap T$ and $Y \cap S$ in the vertex cover. This works because the only uncovered edges would be edges from $X \cap S$ (which we didn't include) to $Y \cap T$ (which we also didn't include). But that edge would be an edge with infinite capacity crossing the cut.

[2] As before, the capacity can also be a large finite number instead of ∞, and actually it can be as low as 1 and things will still work, but it will be easier for us to reason about if it's infinite.