Notes by Samir Khuller.

1 The Min Cost Flow Problem

Today we discuss a generalization of the min weight perfect matching problem, the max flow problem and the shortest path problem called the min cost flow problem. Here you have a flow network, which is basically a directed graph. Each edge (i, j) has a cost c_{ij} and capacity u_{ij}. This is denoted by a tuple (c_{ij}, u_{ij}) for the edge (i, j). Nodes v have supplies/demands $b(v)$. If $b(v) > 0$ then v is a supply vertex with supply value $b(v)$. If $b(v) < 0$ then v is demand vertex with demand $-b(v)$.

Our goal is to find a flow function $f : E \longrightarrow \Re$ such that each vertex has a net outflow of $b(v)$ and the total cost is minimized. As before, the flow on an edge cannot exceed the corresponding capacity. The *cost* of the flow function is defined as $\sum_{e \in E} f(e) \cdot c(e)$. (In other words, supply nodes have a net outflow and demand nodes have a net inflow equal to their $|b(v)|$ value.)

We will assume that the total supply available is exactly equal to the total demand in the network. (In general, for a feasible solution to exist we need to assume that the total supply is at least the total demand. We can add a "dummy" vertex that absorbs the surplus flow, and thus can assume that the total supply and demand values are the same.)

The reader can easily see that the shortest path problem is a very special case when we need to ship one unit of flow from s to t at min cost, where each edge has capacity one. The min weight perfect matching problem is also a special case when each vertex on one side has $b(i) = 1$ and the vertices on the other side have $b(i) = -1$. When all edges have cost zero, it clearly models the max flow problem.

We first discuss an application of this problem to show how powerful it is.

Caterer problem: A caterer needs to provide d_i napkins on each of the next n days. He can buy new napkins at the price of α cents per napkin, or have the dirty napkins laundered. Two types of laundry service are available: regular and express. The regular service requires two working days and costs β cents per napkin, the express service costs γ cents per napkin and requires one working day. The problem is to compute a purchasing/laundring policy at min total cost.

This problem can be formulated as a min cost flow problem and can handle generalizations like inventory costs etc.

The basic idea is to create a source vertex s that acts as a supply node of new napkins. For each day $i, \leq i \leq n$, we create two vertices x_i and y_i. We set $b(x_i) = -d_i$ so this is a demand vertex with the demand equal to the requirement for day i. We set $b(y_i) = d_i$ so this is a supply vertex that can provide the dirty napkins to future days. We set $b(s) = \sum_{i=1}^{n} d_i$. From vertex s there is an edge to each x_i vertex with parameters (α, d_i). From $y_i, 1 \leq i \leq n-2$ there is an edge to x_{i+2} with parameters (γ, d_i). From $y_i, 1 \leq i \leq n-3$ there is an edge to x_{i+3} with parameters (β, d_i). These edges denote the fact that we can launder napkins by using the express service and provide them after a gap of one day at cost γ per napkin, and launder napkins by using the regular service and provide them after a gap of two days at a cost of β per napkin. The total number of such napkins is clearly upper bounded by d_i (number of dirty napkins on day i). Finding a min cost flow, gives us the way to find a min cost solution. A formal proof of this is left to the reader. You should note that we need to create two vertices for each day since one node absorbs the flow (clean napkins) and one provides the flow (used napkins). We also create one sink vertex t that can absorb the surplus napkins at zero cost. (Since the total supply in the network exceeds the demand.)

Residual Graph: We define the concept of residual graphs G_f with respect to a flow function f. The capacity function is the same as before. The cost of the reverse edge is $-c_{ij}$ if the cost of edge (i, j) is c_{ij}. This corresponds to the fact that we can reduce the cost by pushing flow along this edge, since this only reduces the flow that we were pushing in the forward direction.

We now discuss a very simple min cost flow algorithm. We can first compute a feasible solution, by ignoring costs and solving a max flow problem. We create a single source vertex s, add edges from s to each supply node v with capacity $b(v)$. We create a single sink vertex t, and add edges from each demand vertex

v to t with capacity $|b(v)|$. If the max flow does not saturate all the edges coming out of the source then there is no feasible solution.

Unfortunately, this solution may have a very high cost! We now show how to reduce the cost of the max flow. Construct the residual graph, and check to see if this has a negative cost cycle (one can use a Bellman-Ford shortest path algorithm for detecting negative cost cycles). If it does have a negative cost cycle we can push flow around the cycle until we cannot push any more flow on it. Pushing a flow of δ units around the cycle reduces the total flow cost by $\delta|C|$ where C is the cost of the cycle. If there is no negative cost cycle in G_f then we can prove that f is an optimal solution. Notice that pushing flow around a cycle does not change the net incoming flow to any vertex, so all the demands are still satisfied. We only reduce the cost of the solution.

The next question is: how do we pick a cycle around which to cancel flow? Goldberg and Tarjan proved that picking a min mean cycle[1] gives a polynomial bound on the total number of iterations of the algorithm. One can also try to find the cycle with the least cost, but then one might only be able to push a small amount of flow around this cycle. One could also try to identify the cycle which would reduce the cost by the maximum possible amount.

Lemma 1.1 *A flow f is a min cost flow if and only if G_f has no negative cost cycles in it.*

Proof:

If G_f has a negative cost cycle, we can reduce the cost of flow f, hence it was not an optimal flow. If G_f has no negative cost cycles, we need to argue that f is an optimal solution. Suppose f is not optimal, then there is a flow f^* that has the same flow value, but at lower cost. Consider the flow $f^* - f$; this is a set of cycles in G_f. Hence f can be converted to f^* by pushing flow around this set of cycles. Since these cycles are non-negative, this would only increase the cost! Hence $c(f) \leq c(f^*)$. Since f^* is an optimal flow, the two must have the same cost. $\square$

In the next lecture we will study a different min cost flow algorithm.

[1] A min mean cycle minimizes the ratio of the cost of the cycle to the number of edges on the cycle.