Advanced Algorithms

Class Notes for Monday, October 29, 2012

Bernard Moret

1 Stable Matching: A solution and some proofs

We saw the definition of this problem last lecture. Now let us look at a solution of the problem: an algorithm that is guaranteed to return a stable matching and thus at once proves that such matchings always exist and gives us a way to produce one. The algorithm is one used in polite society in western cultures for much of the past three centuries, also characterized by the rather misleading statement "man proposes, woman disposes."

More precisely, our algorithm proceeds in *rounds*, during which a man or woman can be free or engaged. Once every person is engaged, a stable matching has been obtained and the engagements can be turned into marriages. Initially, every man proposes to the woman at the top of his list; more generally, at each round every free man proposes to the next best woman on his list (whether or not she is free). At each round, a woman will consider the man (if any) she is engaged to from a previous round, as well as any proposals she has received, and pick the best of the collection as her new engagement. (In this process, if a woman becomes engaged to a new man, a previously free man becomes engaged and a previously engaged man becomes free.) A man who is rejected by the ith woman on his list (immediately or because she has just received a better proposal) will propose in the next round to the $(i+1)$st woman on his list. We use as many rounds as needed to complete the process, that is, to reach a state in which every man and woman is engaged.

First, note that this process must converge, since no man can propose to the same woman twice and a woman must always accept the best proposal she has received. In fact, you can verify that the number of rounds for n men and n women cannot exceed n^2-2n+2—the worst case arises when a man engaged to his top choice on the first round gets dislodged $n-1$ rounds later when this woman gets a proposal she prefers, at which point the man becomes free again, proposes to his second choice, gets engaged to her, but then gets dislodged $n-2$ rounds later when this second woman gets a proposal she prefers, and so on.

Second, we verify that the resulting marriage is stable. Suppose John prefers Mary to his own wife in the final matching. Then John must have proposed to Mary before he proposed to his own wife and have been rejected by Mary. Thus Mary was engaged, or later became engaged, to someone she prefers to John; as further changes in Mary's choice of fiancé can only occur when she receives an even better proposal, she clearly ranked her husband higher than she did John, and so the marriage is stable.

Third, probably to no woman's surprise, we claim that the resulting stable matching is man-optimal, in the sense that, in any stable matching (if there exist several), no man can do end up with a higher-ranked woman than in the matching produced by the algorithm. This

result we prove by induction on the rounds of the algorithm. Consider the first time that any woman rejects a proposal during the running of the algorithm (clearly, if no proposal is rejected, then each man got his first choice and the matching is man-optimal). If this woman, call her Mary, rejects some man John, it is because she received a proposal from some other man she prefers, say Nigel. We must show that John cannot marry Mary without causing instability—that there is no stable solution in which John is married to Mary. By induction, Mary is the best possible woman for Nigel at this stage—and any further choices for Nigel, by the nature of the algorithm, will be farther down his list, so Nigel prefers Mary to all possible women he could marry in a stable matching. Now, if John were to marry Mary, Nigel would have to be married to one of these other possible women and would prefer Mary to his own wife; we have already assumed that Mary prefers Nigel to John, so a marriage of John and Mary would create an instability. Thus the only rejections men get in this algorithm are from women to whom they cannot be married in any stable matching: the algorithm produces a man-optimal matching. (It is easily verified that this matching is also woman-pessimal, in the sense that, in any possible stable matching, a woman ends up with a man ranked at least as high as the one she ends up with in the man-optimal matching. Society, run by men for centuries (if not millennia) had devised a procedure that, under the guise of courtly behavior and behind the misleading screen of "woman disposes," ensured that men always go the best possible outcome in marriage choices. Curiously, it is extremely difficult to devise an algorithm that produces a stable matching that is fair to both sexes—it is not even all that simple to produce a quantitative definition of "fairness" in this context.

The same problem run for, say dorm room assignments among a collection of students (each student ranks all others as potential roommates) is not solvable in this manner, because stable solutions need not exist for all instances. The same problem run for tripartite marriages (imagine some extraterrestrial races of beings with three genders who also form marriages with a single individual of each gender) is NP-hard to solve.

2 Network Flow

We now turn our attention to the other major family of optimization problems solvable exactly by iterative improvement methods, the family of network flow problems.

2.1 The problem

A network is just a graph $G = (V,E)$, but one equipped with two special vertices, a source $s \in V$ and a sink $t \in V$, and with a (positive integer) capacity function $c\colon E \longrightarrow \mathcal{N}^+$. The goal is to compute a maximal flow on the network, where a flow is an assignment of integer values (positive or negative) to edges, $f\colon E \longrightarrow \mathcal{Z}$, subject to capacity constraints, i.e., for each edge $e \in E$, we must have $|f(e)| \leq c(e)$, and to flow conservation constraints, that is, at each vertex $v \in V$ other than the source and sink, we must have $\sum_{(u,v)\in E} f(u,v) = 0$. (Flow is oriented: it moves in a specific direction along an edge, hence the zero sum in the last sentence, indicating that the sum of the flows coming into vertex v equals that of the

flows leaving that vertex.) If flow through an edge equals the capacity of that edge, we say that the edge is *saturated*. The value of a flow is the sum of the edge flows out of the source or, equivalently, the sum of the edge flows into the sink. The maxflow problem asks for the maximum flow on the network. In order to prove some basic results about flows and maximum flows, we first need a few definitions.

2.2 Cuts, residual graphs, and augmenting paths

A cut in a network $G = (V, E)$ is a bipartition (a partition into two subsets) of the vertices of the graph, $V = \{V_1.V_2\}$. with $s \in V_1$ and $t \in V_2$. An edge $\{u, v\} \in E$ is *cut* by the bipartition whenever we have either $u \in V_1$ and $v \in V_2$ or $u \in V_2$ and $v \in V_1$. The *capacity* of a cut is the sum of the capacities of the edges of the network that it cuts. A cut of minimum capacity is called simply a *minimum cut*. Notice that, in order for a flow to go from s to t, it must go "through the cut," that is, all of it must go through the cut edges from the side of the s to the side of t. (You can imagine drawing the network in the plane and tracing a curve that goes through every cut edge, forming a boundary between the two sides of the bipartition: the flow must cross the boundary.) Thus it follows that any flow from s to t cannot exceed the capacity of any cut; in particular, the value of a maximum flow from s to t is upper bounded by the capacity of the minimum cut. We shall show in the next subsection that the value of a maximum flow in fact equals the capacity of the minimum cut, but first we need to consider the effect of any flow on the structure of the network.

We begin by noting that a purely greedy approach to the problem cannot work: if we simply attempt to push out of the source as much flow as the combined capacity of all edges incident on the source, we are very likely trying to push too much flow into the network: after all, the collection of edges incident to the source is itself a cut (corresponding to the bipartition $V_1 = \{s\}$ and $V_2 = E - \{s\}$) and need not be a minimum cut. It is pretty obvious from the outset that we need to focus on paths from s to t. Consider such a path in the original network: as each of its edges has some capacity, it is clear that the value of the flow we can push along this path is at most the capacity of the edge of smallest capacity. Since all capacities are strictly positive integers, any path allows us to push at least one unit of flow. Let the path consist of vertices $s, v_1, \ldots, v_{k-1}, t$ with its k edges denoted $e_1, e_2, \ldots, e_k$ of capacities $c_1, c_2,]ldots, c_k$; and say that edge e_i has minimum capacity along this path. Then we can push a flow of c_i units along the path; this will change all forward capacities along the path from c_j to $c'_j = c_j - c_i$, $1 \leq j \leq k$, causing the capacity of the ith edge to drop to zero and thus making the path no longer usable. We could then formulate a first (and erroneous, as we shall see) algorithm to compute a flow using successive augmenting paths as follows:

- Initialize the flow to zero: $f(v, w) \leftarrow 0$ for all v and w.
- Repeat these steps until there is no path from s to t:
  - Find any path, P, from s to t in the current graph; let Δ be the capacity of the arc of minimum capacity along this path, $\Delta = \min_{(v,w)\in P} c(v, w)$.

- Record the change to the flow, i.e., let $f(v,w) \leftarrow f(v,w) + \Delta$ for all arcs of P and maintain skew-symmetry.
- Decrease the capacity of all arcs along P by Δ, deleting all arcs for which the capacity becomes zero.

Unfortunately, it is not hard to devise very simple networks of a few nodes on which this strategy fails. The problem is that it has a simplistic view of what constitutes an augmenting path because it is too closely attached to the original graph. We already observed that an edge of capacity c can admit flow of up to c units in *either direction*. We used an undirected graph in the problem statement because it leads to a simple formulation of the problem, of cuts, and other static aspects of the network, but once a first flow is imposed on the network we need to switch to a directed graph representation, with arcs in lieu of edges. In the initial network with no flow, an edge $\{u,v\}$ of capacity c really stands for two arcs, (u,v), and (v,u), each of capacity c. If we now push Δ units of flow from u to v, then the capacity of the forward arc, (u,v), decreases by Δ, but that of the reverse arc, (v,u), *increases* by that same amount. (Intuitively, the capacity is higher, because, in order to saturate arc (v,u) with c units of flow, we must first push Δ units to cancel the existing flow, then push the c units for saturation.)

Thus we define a *residual graph* for a network $G = (V,E)$ with respect to flow f as a directed graph $R_f(G) = (V,A)$, where, if $\{u,v\}$ is an edge of G with capacity c, then $(u.v)$ is an arc of $R_f(G)$ with capacity $c - f(u,v)$ and (v,u) is an arc of $R_f(G)$ with capacity $c - f(v,u) = c + f(u,v)$—except that, to simplify notation, we remove from $R_f(G)$ any arc thus defined whose capacity is zero. Now we can search for a path from s to t in $R_f(G)$ and use it to augment the flow as described above—and the paths we may now find include paths we could not find before, because our first attempt insisted on pushing flows always in the same direction, whereas, by using residual networks, we can have successive augmenting paths that use the same edge in opposite directions. As we shall see, this notion of augmenting path now suffices to ensure optimality.

2.3 The min-cut max-flow theorem

We are now ready to prove the maxflow-mincut theorem. In its original flavor, this was a purely mathematical result, establishing the existence of certain objects and properties, but giving no clue on how to construct such objects. We state the theorem with an additional statement about augmenting paths; this statement simplifies the proof and enhances the value of the theorem by giving us an algorithmic insight about the maxflow.

Theorem 1. *Let $G = (V,E)$ be a graph with capacity function c and let f be a flow. Then the following are equivalent:*

1. *f is a maximum flow.*
2. *There is no augmenting path in G with respect to f.*
3. *The value of f equals the capacity of some cut.*

Proof. To prove an equivalence, we prove a circular chain of implications.

$(1 \Rightarrow 2)$ We have seen how to augment a flow using an augmenting path.

$(2 \Rightarrow 3)$ If there is no augmenting path in G with respect to f, then the set of vertices, X, reachable from s in the residual graph does not contain t, so that the partition, $V = (X, \overline{X})$, is a cut. The capacity of this cut is

$$\sum_{v \in X, w \in \overline{X}} c(v,w).$$

We argue that this sum is exactly equal to the value of the flow. Our earlier remarks imply that the value of the flow, the amount of material leaving the source, is equal to the flow across the cut from X to $\overline{X}$, after accounting for any backward flow from $\overline{X}$ to X. Due to the nature of this particular cut, there cannot be any backward flow from a vertex $w \in \overline{X}$ to a vertex $v \in X$; if there were, then the residual graph would have an arc from v to w, w would be reachable from s, and w would be in X rather than in $\overline{X}$. Similarly, every arc from X to $\overline{X}$ is saturated, that is, $f(v,w) = c(v,w)$, and the claim is proved.

$(3 \Rightarrow 1)$ Since the capacity of any cut is an upper bound on the value of any flow, it follows that an equality implies that f is a maximum flow.

□

As promised, part (ii), the additional statement about augmenting paths, gives us an algorithmic insight: we can produce a maximum flow by repeatedly finding and using augmenting paths, until no such path remains. Notice also that it verifies an unsupported claim we made earlier, namely that, given integer capacities, there always exists a maximum flow with integer values, since our augmenting paths, starting from integer values, can only produce new integer values.

2.4 Simple algorithms

The first algorithm for network flow was given by Ford and Fulkerson and is the one just suggested: while there remains an augmenting path in the residual network, use this path to augment the flow and update the residual network. Applied to network with integral capacities, this algorithm produces an optimal solution, but it may be quite slow—in fact, it could take exponential time! The problem is not in the spirit of the algorithm, but in the choice of augmenting paths. Consider a very simple network of four nodes: s, t, A, and B, and five edges, from s to A and B, from A to B, and from t to A and B; assign unit capacity to the edge between A and B and some large capacity value M to the other four edges. The maxflow clearly has value $2M$, using the two paths $s-A-t$ and $s-B-t$. However, if, for whatever reason, the augmenting path first discovered is the path $s-A-B-t$, then this path has min capacity of 1 and so the first iteration augments the flow value by 1. After this augmentation, the residual graph has no arc from A to B, but has an arc from B to A

of residual capacity 2. If the next augmenting path selected is $s-B-A-t$, of minimum capacity 2, the flow value increases by 2 and the next residual graph has an arc from A to B of residual capacity 2, but no arc from B to A. In this new residual graph we can choose the augmenting path $s-A-B-t$, again augmenting by 2, and so forth. Overall, it will take M iterations to reach the maximum flow of $2M$ units, with successive augmenting paths pushing flow between A and B in alternative directions. (Of course, if we had chosen the augmenting paths $s-A-t$ and $s-B-t$, we would have augmented by M each time and so been done in two iterations.) M is not polynomial in the input size, which only includes a factor of $\log_2 M$.

So what is needed is to choose the augmenting paths more carefully. The first and simplest solution is, by analogy with our maximum matching algorithm, to choose the *shortest* augmenting path. Results very similar to those we obtained with matching can now be established: for instance, successive shortest augmenting paths are of nondecreasing length. We could then establish that it takes $O(|V|\cdot|E|)$ augmentations to produce a maximum flow; since each iteration search for a path from s to t, each iteration takes $O(|E|)$ time and so we have a simple algorithm that runs in $O(|V|\cdot|E|^2)$ time. This is not very fast, so we could next do what we did with maximum matching: use a single search from s to t to identify as many disjoint augmenting paths as possible. Finding and using such a collection of augmenting paths at each iteration will cut all paths from s to t in the residual graph (the one in which the search is conducted: after augmenting the flow, new paths from s to t may be created). These methods are consequently called *blocking flow* methods and we can prove that at most $|V|-1$ successive iterations (each producing a blocking flow) will be needed to produce a maximum flow—but each iteration is not just a simple BFS and now takes $O(|V|\cdot|E|)$ time; as a result blocking-flow methods run in $O(|V|^2\cdot|E|)$ time (Dinic' algorithm). This is an improvement, but perhaps not much of one: we can also prove a theorem stating that there always exists a sequence of at most $|E|$ augmenting paths that will produce the maximum flow from a null flow—suggesting that an ideal algorithm might be able to run in $\Theta(|V|\cdot|E|)$ time.

However, all of these approaches have been superseded by the so-called Push-Relabel (or Gravity Flow) approach, so we will not prove any of the results mentioned above. The Push-Relabel algorithm is based on our first greedy idea: push as much flow "down into" the source as possible—that is, saturate every edge out of the source. This, of course, makes two (generally false) assumptions: first, that in a maximum flow, every edge incident to the source is used to move flow away from the source; and second, that every such edge is saturated. The saturation is part of the greedy strategy: push as much as you can down an edge. The first assumption is a bit more subtle: it works from the idea of some type of ordering of vertices—imagine using water for flow and flexible tubes for edges, then you can visualize placing the source high, the sink low, and pouring water into the source—as much as can flow through the tubes without overflowing. The relative position of the nodes dictates the direction of flow in each tube and this may have to be adjusted many times before it corresponds to the correct directions of flow in an optimal solution—this is the reason for the second part of the name, "Relabel", i.e., change the relative height of vertices in such an arrangement. We shall study this approach in detail in the next lecture.