CSE 202 NOTES FOR NOVEMBER 21, 2002

Maximal Bipartite Matching

We consider a new problem to introduce the notion of reduction and to describe the usefulness of network flow.

Instance: An undirected graph $G = (V, E)$. Vertices in V can be divided into two disjoint subsets L and R, such that

$$e = (u, v) \in E \implies (u \in L \wedge v \in R) \otimes (u \in R \wedge v \in L).$$

Solution Format: A subset M of E.
Constraints: Any node $v \in V$ may have only one edge incident to it in the output set M.
Objective: Maximize $|M|$.

An example of an $m \times n$ bipartite graph is shown in figure 1. Here $m > n$.

Our solution to this is fairly straightforward: create nodes s and t to the left of L and right of R respectively, and connect each node in L to s and each node in R to t. Set the "capacity" associated with every edge in this new graph to 1, and use the Ford-Fulkerson method on the graph to determine the matching. Because we're using Ford-Fulkerson, we know that the matching gives us an integral flow since the edges are all of integer capacity.

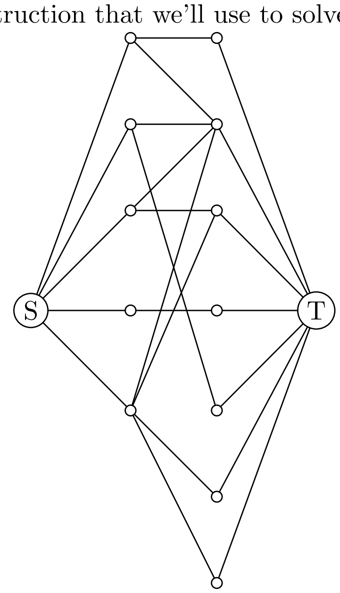

Figure 1. A sample bipartite graph. Nodes s and t (and the edges incident to them) are not actually part of the bipartite graph; they are simply a construction that we'll use to solve the problem.

How can we prove that this approach is correct? Let f be any integer-value flow, and let M_f be the set of edges from E (not including the augmenting edges to connect to s and t, which are artificial constructs) such that $f(e) = 1$. Why is M_f a matching? Because $(u, v_1) \in M_f \implies \nexists v_2$ st $(u, v_2) \in M_f$, and similarly for u. This particular case would cause flow into an vertex to be 2, but the flow in would be 1, which can't happen at nodes-that-aren't-endpoints.

Now that we've shown that we have a matching, can we show that it is optimal? We know that $|\text{Flow}| \geq |M_f|$ because each edge used from s in the flow feeds into an edge in M_f and vice versa, since all the weights are unity. We also need to show that the maximum matching is $\leq$ the maximum flow; construct a flow from a matching, and the same reasoning applies in the reverse direction. For every $e = (u, v) \in E$, send one unit from $s \to u$ and from $r \to t$. Because we have a maximal match, we must have a maximal flow; this was rushed a bit, because of other topics to cover.

Reductions

A problem reduction is an elegant construct that allows one to frame a particular computational problem in an alternative light. In general, one shows an algorithmic transformation from a problem who's solution is unknown to a problem that can be solved easily. There are some mathematical requirements to a reduction which we will cover shortly. Unlike the reductions in complexity theory, one has to be concerned with the execution time of the reduction—while a reduction may be theoretically possible, if it requires an order of n^{100} or 2^n to accomplish the transformation, then it is hardly a practical technique for solving the problem.

The previous example posed an interesting problem: if we reduce optimization problem 1 to optimization problem 2, how do we show that our reduction is valid and that a solution to 2 implies a solution to 1.

Following the basic idea of how we have specified problems so far, we really need to show that a mapping exists from an instance of the first problem to the instance of the second problem (call this mapping f). We also need to show that a solution for 2 is also a solution for 1, and vice versa (call these g and h). We need g to determine how long it will take to translate the answer back into a form suitable for problem 1, and we need an h to guarantee that a solution for the first implies that a solution exists in the second problem. That is, we're showing that an optimal solution in 2 is also an optimal solution in 1, and vice versa; if this is not true, then the reduction doesn't work because we might miss the solution to 1 if it doesn't map to an optimal solution in 2 (our algorithm for 2, after all, finds the optimal solution for *2*.

Approximation Ratios and Reductions

In approximation algorithm reductions, one has to be concerned with breaking any approximation ratio guarantees during the problem transformation. The reason that this is a concern is that there are several pieces involved in a reduction, and not all of them are well-behaved. For example, it may be that, in reducing a problem Π to a problem Π' we find that some solution to an instance of Π is really not a solution to the corresponding instance of Π'. This would imply that, for some instances of Π, Π' is not a reasonable reduction. Of course, this was a concern when reducing exact algorithms too, but we now need to worry about the following case: if it turns out that some solution to Π is $(1/10)OPT$ in Π' while all other solutions to instances are $(1/2)OPT$, then our approximation algorithm can only be guaranteed to give an AR of $1/10$. Despite these caveats, there is a certain class of reductions which are well-behaved even for approximation algorithms, and we call these (α, β)-preserving reductions.

We will review here the basics behind problem reductions, and then briefly introduce the notion of (α, β) preserving reductions.

In the following, let Π and Π' be problems (or rather, the set of all instances of two particular problems). Furthermore, let Π' be *well-characterized*, in the sense that an exact algorithmic solution exists for all $\pi' \in \Pi'$. Suppose that Π does not have this property—some, or all of its instances are not solvable by any known algorithmic means. In the following, let $S(\Pi)$ be the set of solutions for instances of a problem Π. A *reduction* from Π to Π' consists of three functions, $f : \Pi \to \Pi'$, $g : S(\Pi') \times \Pi \to S(\Pi)$, and $h : S(\Pi) \times \Pi' \to S(\Pi')$. Perhaps a more informative way of putting this is in a picture:

$$\begin{array}{rcl} \pi \in \Pi & \to^{f(\pi)} & \pi' \in \Pi' \\ s \in S(\Pi) & \leftarrow^{g(s',\pi)} & s' \in S(\Pi') \\ t \in S(\Pi) & \to^{h(s,\pi')} & t' \in S(\Pi') \end{array}$$

Furthermore, we must have the guarantee that:

$$\begin{array}{rcl} \mathrm{Val}_\Pi(s) > \mathrm{Val}_\Pi(t) & \Longrightarrow & \mathrm{Val}_{\Pi'}(s') > \mathrm{Val}_{\Pi'}(t') \\ & & s' = h(s, \pi') \\ & & t' = h(t, \pi') \\ \mathrm{Val}_{\Pi'}(s') > \mathrm{Val}_{\Pi'}(t') & \Longrightarrow & \mathrm{Val}_\Pi(s) > \mathrm{Val}_\Pi(t) \\ & & s = g(s', \pi) \\ & & t = g(t', \pi) \end{array}$$

In words, these inequalities are stating that if s' is an optimal solution for π', then $s = g(s', \pi)$ is an optimal solution for π. If s were not optimal, then there would be another solution t' in π' that had a better value than s', which is clearly impossible. A similar line of reasoning holds for optimal solutions in π as applied to π'.

The generalization of this algorithm design technique to approximation algorithms relies on changing these two inequalities to the following:

$$\begin{array}{rcl} \mathrm{Val}_\Pi(s) > \mathrm{Val}_\Pi(t) & \Longrightarrow & \mathrm{Val}_{\Pi'}(s') > \alpha \mathrm{Val}_{\Pi'}(t') \\ & & s' = h(s, \pi') \\ & & t' = h(t, \pi') \\ \mathrm{Val}_{\Pi'}(s') > \mathrm{Val}_{\Pi'}(t') & \Longrightarrow & \mathrm{Val}_\Pi(s) > \beta \mathrm{Val}_\Pi(t) \\ & & s = g(s', \pi) \\ & & t = g(t', \pi) \end{array}$$

In other words, if s is a better solution than t for some instance π, then the corresponding s' is some factor α better than the corresponding t'. The converse must also be true, but note that there may be a different factor β involved in the opposite direction: this means that the approximation error introduced *by the* *reduction* depends on the direction, regardless of whether or not you use the h function in your actual algorithm.

If both of the above inequalities are true, then we can say that our approximation algorithm is preserved across reduction, or that this is an (α, β)-preserving reduction. Not surprisingly, the adjustment to the approximation ratio in such a reduction is $\alpha\beta$.

A reduction, then, is a triplet of functions (f, g, h) with the following signatures:

$$\begin{array}{rcl} \pi \in \Pi & \rightarrow^f & \pi' \in \Pi' \\ s \in S(\Pi) & \leftarrow^g & s' \in S(\Pi') \\ t \in S(\Pi) & \rightarrow^h & t' \in S(\Pi') \end{array}$$

We call a reduction (α, β)-preserving if we can also ensure that the following two inequalities hold for all solutions in $S(\Pi)$ and $S(\Pi')$ respectively:

$$\begin{array}{rcl} \mathrm{Cost}_{\Pi}(g(s')) & \leq & \alpha \mathrm{Cost}_{\Pi'}(s') \\ \mathrm{Cost}_{\Pi'}(h(s)) & \leq & \beta \mathrm{Cost}_{\Pi}(s) \end{array}$$

Lemma 0.1. *If A is an algorithm for Π' that achieves approximation ratio δ and (f, g, h) is an (α, β)-preserving reduction, then $g(A(f(\pi)))$ achieves approximation ratio $\alpha\beta\delta$.*

Proof: Let $\pi' = f(\pi)$. Let OPT' be an optimal solution to π'. Since $\mathrm{Cost}_{\pi'}(A(\pi')) \leq \delta \mathrm{Cost}_{\pi'}(OPT')$, we can say that $\mathrm{Cost}_{\pi}(g(A(\pi))) \leq \alpha\delta \mathrm{Cost}_{\pi}(OPT')$ by the first inequality combined with the assumed optimality of OPT'. Furthermore, let OPT be an optimal solution to π. We know that $\mathrm{Cost}_{\Pi'}(OPT') \leq \mathrm{Cost}_{\Pi'}(h(OPT))$ (again, by the assumed optimality of OPT' for π'). We know further that $\mathrm{Cost}_{\Pi'}(h(OPT)) \leq \beta \mathrm{Cost}_{\Pi}(OPT)$ by the second inequality above. Thus, $\mathrm{Cost}_{\Pi}(g(A(\pi'))) \leq \alpha\delta(\beta \mathrm{Cost}_{\Pi}(OPT))$ as desired. □