CS 161 - Design and Analysis of Algorithms (Summer 2012)
Stanford University

Solutions to Final Review
August 9, 2012

Problem 1 (20 Points). There are many common variations of the maximum flow problem. Here are four of them:

(a) There are many sources and many sinks, and we wish to maximize the total flow from all sources to all sinks.

(b) Each *vertex* has a capacity on the maximum flow that can enter it.

(c) Each edge has not only a capacity, but also a *lower bound* on the flow it must carry.

(d) The outgoing flow from each node v is not the same as the incoming flow, but is smaller by a factor of $(1 - \epsilon_v)$, where ϵ_v is a loss coefficient associated with node v. In this case, we wish to maximize the flow to the sink.

Each of these can be solved efficiently. Show this by reducing (a) and (b) to the original max-flow problem, and reducing (c) and (d) to linear programming.

Solution:

(a) At a new source s with a directed edge of infinite capacity to each of the original sources, and a new target t with a directed edge of infinite capacity from each of the original sources. Any flow from s to t in this graph corresponds to a flow in the original graph from the sources to sinks, and both flows have the same size. Thus, the maximum flow in the new graph equals the maximum flow in the original graph.

(b) Replace a node$v$ with two nodes v_{in} and v_{out}. Any edges into v now go to v_{in}, and any edges out of v now leave through v_{out}. Additionally, there is a single edge form v_{in} to v_{out} whose capacity is the capacity of the node v. A flow in the old graph corresponds exactly to a flow in the new graph: if there is a flow f on edge (u, v), there is a flow f along edge $(u_{out}, v_{in}$. The capacity of the edge (v_{in}, v_{out}) enforces the constraint that the flow through v is bounded.

(c) Modify the LP constraints so that $c(e) \leq f_e \leq w(e)$ where $c(e)$ is the lower bound for the edge

(d) Instead of having the constraint flow in $=$ flow out for every node, we have flow in $(1 - \epsilon_v) =$ flow out. We also change the objective function so that we are maximizing the sum of the flows into the sink

Problem 2 (20 Points). A *vertex cover* of an undirected graph $G = (V, E)$ is a subset of the nodes which touches every edge — that is, a subset $S \subset V$ such that for each edge $(u, v) \in E$, one or both of u, v are in S.

Show that the problem of finding the minimum vertex cover in a *bipartite* graph reduces to maximum flow. *Hint: Can you relate this problem to the minimum cut in an appropriate flow network*

Solution: Let (A, B) be the sets of nodes such that all edges have one endpoint in A and one in B.

Construct the same flow network as for bipartite matching: Create new nodes s and t, and have an edge from s to each node in A, and edge from each node in B to t, and direct all edges in G from A to B. All edges have weight 1.

Let C b a cut in this new graph. Without loss of generality, we can assume C does not contain any of the edges between A and B (if an edge (a, b) was in the cut, we can take that edge out of the cut, and add the edge (s, a) to the cut. If we had a cut before, we surely still have a cut now). Therefore, every edge in C is either (s, a) for some $a \in A$, or (b, t) for some $b \in B$. Let S be the union of the a nodes and b nodes that are the endpoints of edges in C. Then $|S| = |C|$. Further, suppose there is an edge(a, b) in G whose endpoints are not in S. But then there is a path $s - a - b - t$, with none of the edges in C, which is impossible if C is a cut. Therefore, S is a vertex cover.

Conversely, suppose there is a vertex cover S. Let C be the set of edges that either go from s to a node in S, or from a node in S to t. Clearly $|C| = |S|$. Now, suppose there is some path $s - a - b - t$. If (s, a) is not in C, then $a \notin S$, so b must be in S since (a, b) is covered by S. But then $(b, t) \in C$. Thus, after removing the edges in C, there is no path from s to t, meaning C is a cut.

Thus, if we find a minimum cut, we can produce a vertex cover, and this vertex cover is minimal (since any smaller vertex cover would yield a smaller cut).

Problem 3 (20 Points). Suppose we have a bipartite matching problem with n boy and n girls. Hall's Theorem says that there is a perfect matching if and only if the following condition holds: any subset S of boys is connected to at least $|S|$ girls.

Prove this theorem. *Hint: the max-flow min-cut theorem should be helpful.*

Solution: If there is a perfect matching, it means that the maximum flow in the flow network has size n. By the max-flow min cut theorem, this means the minimum cut has size n. Now, suppose there is some subset of boys S connected to fewer than $|S|$ girls. We describe a cut of the flow network: for every boy not in S, include the edge from the source node to that boy. For every girl connected to a boy in S, also cut the edge from that girl to the target node. If we run a DFS starting at the source, we can only visit the nodes in S, since the edges to the other boys are in the cut. Once we visit one of the boys in S, we must visit one of the girls that boy connects to, but then we cannot get to the destination because the edge from girl to destination is in the cut. Therefore, the DFS never gets to the destination, so this is indeed a cut. But the size of the cut is $n - |S|$ for the edges from source to boys not in S, plus fewer than $|S|$ edges from the girls. Therefore, there are fewer than n edges in the cut, a contradiction.

If there is not a perfect matching, then there is a cut of size less than n. Let S be the set of boys where the edge from source to that boy is not in the cut. There are $n - |S|$ edges in the cut from source to the other boys. Further, there must be fewer than $|S|$ other edges in the cut. Now, suppose S connects to at least $|S|$ girls. It is then impossible to disconnect the boys in S from the source using fewer than $|S|$ edges. Therefore, S connects to fewer than $|S|$ girls.

Problem 4 (10 Points). Show that for any problem in NP, there is an algorithm which solves it in time $O(2^{p(n)})$, where n is the size of the input instance and $p(n)$ is a polynomial (which may depend on the problem).

Solution: Recall that for any NP problem, we have a binary relation $R(x, y)$, where R is computable in time polynomial in the size of x. Since the running time of R is only polynomial in the size of x, it can read only a polynomial number of bits of y, say $q(n)$. We then have the following algorithm to solve the problem: iterate over all $2^{q(n)}$ different y, and see if $R(x, y) = True$. If it does for some y, then output $True$. Otherwise output $False$. We evaluate R $2^{q(n)}$ times, and evaluating R takes, say $r(n)$ time for some polynomial $r(n)$. Therefore, the total running time is $O(r(n)2^{q(n)}) = O(2^{q(n)\log r(n)}) \leq O(2^{q(n)r(n)})$

Problem 5 (30 Points). In the `NODE-DISJOINT PATHS` problem, the input is an undirected graph in which some nodes have been specially marked: a certain number of "sources" $s_1, ..., s_k$ and an equal number of "destinations" $t_1, ..., t_k$. The goal is to find k node-disjoint paths (that is, paths which have no nodes in common) where the ith path goes from s_i to t_i. Show that that this problem is NP-Complete.

Here is a sequence of progressively stronger hints:

i. Reduce from `3SAT`

ii. For a `3SAT` formula with m clauses and n variables, use $k = m + n$ sources and destinations. Introduce one source/destination pair(s_x, t_x) for each variable x and one pair (s_c, t_c) for each clause c.

iii. For each `3SAT` clause, introduce 6 new intermediate nodes, one for each literal occurring in that clause and one for its complement

iv. Notice that if the path from s_c to t_c goes through some intermediate node representing, say, an occurrence of variable x, then no other path can go through that node. What node would you like the other path to be forced to go through instead.

Solution: We follow the hints. For each variable x, introduce a source/destination pair (s_x, t_x), and for each clause c introduce a source/destination pair (s_c, t_c). . For every clause c, we also introduce 6 other nodes, one for each literal in the clause, and one for the compliment. For each literal ℓ in c, create the path $s_c - \ell - \overline{\ell} - t_c$.

Additionally, create a path from each s_x through all the xs that are next to t_c nodes, finishing at t_x, as well as another path through all the $\overline{x}$s that are next to t_c nodes.

If there is a satisfying assignment, we can get a node-disjoint path for every source target pair. Connect s_x to t_x through the path that goes through xs if x is true, or $\overline{x}$s if x is false. Additionally, since every clause is satisfied, every clause c has a literal ℓ that is true. Choose the path $s_c - \ell - \overline{\ell} - t_c$. Notice that since ℓ is true, no other path goes through the $\overline{\ell}$ node, so all the paths are disjoint.

If all of the source/target pairs have node disjoint paths, set each variable x corresponding to which path is taken from s_x to t_x. Since there is some path from s_c to t_c, it must go $s_c - \ell - \overline{\ell} - t_c$ for some literal ℓ. But this means no path went through $\overline{\ell}$, meaning ℓ must be set to true, so the clause c is satisfied. Hence, this assignment is a satisfying assignment.

Total points: 100