CSOR W4231 (sec. 001, 002, H02): Analysis of Algorithms Nov 21

Homework 6 Solutions

Instructors: *Alex Andoni, Cliff Stein*

Instructions. Please follow the homework policy and submission instructions in the Course Information handout. A few highlights to remember:

- follow the collaboration and academic honesty policies;
- write your name, uni, and collaborators on the top page;
- submission is via GradeScope (you are encouraged to use LaTeX, using the provided template);
- if you don't know how to solve a particular part of a problem, just write "*empty*", for which you will get 20% of the points of that part (in contrast, note that non-sensical text may get 0%).

Note that, for each bullet item of a problem, you can use the previous bullets as "black box", even if you didn't manage to solve them. Similarly, you can use anything from the lectures as black-box.

Problem 1

Let G be a graph and f be a maximum flow, which is *acyclic* (no unit of flow ever comes back to a node already visited). Give an algorithm that outputs a series of at most $|E|$ augmenting paths that, when augmented along would give rise to the flow f.

Note that you are not asked to give a new maximum flow algorithm, you asked how, given the maximum flow f, you can recreate a series of augmenting paths.

Solution:

Let f be a maximum flow on G. Either $f(e) = 0 \ \forall e \in E$ or there is some edge e such that $f(e) = \min\{f(d) : f(d) > 0\}$. There is some augmenting path p that contains e (or else f is not a flow). Augmenting along p results in a new graph G' with maximum flow f'. f' is 0 on one more edge than is f. Thus after $|E|$ such augmentations the resulting graph can no longer be augmented, and the algorithm terminates.

Formally, we repeat the following process: find a path with non-zero flow from s to t, which must always exist if the final flow is non-zero. Reduce the flow on this path until some edge becomes 0, which will the minimum flow edge on the path, and then add this path to our set of outputted augmenting paths. We repeat this until there are no paths from s to t with non-zero flow. This can be implemented using a DFS to iterate over all paths from s to t.

Problem 2

Suppose we want to solve the $s-t$ shortest path in a weighted directed graph G with the extra condition that the number of edges must be at most k, where k is the smallest number of edges in any path from s to t (ie, length of the shortest path in the unweighted version of the graph). You are to solve this problem by formulating it as an LP. You can assume that all weights are positive.

a) For a given graph $G = (V, E, w)$, nodes s, t, and a bound k, formulate the problem of finding the $s - t$ shortest path with at most k edges as an LP program with $e = |E|$ variables. *Hint:* first think of how you would formulate the program if you could set the unknowns x_i to be from $\{0, 1\}$, and then relax to $0 \leq x_i \leq 1$.

Solution: We will formulate our LP with a variable f_{uv} for each edge (u, v) in the original graph G.

$$\begin{aligned}
\min_{f} \quad & \sum_{(u,v)\in E} w_{uv} f_{uv} \\
\text{s.t.} \quad & \sum_{(s,v)\in E} f_{sv} - \sum_{(v,s)\in E} f_{vs} = 1 \\
& \sum_{(u,v)\in E: u\neq s,t} f_{uv} - \sum_{(v,u)\in E: u\neq s,t} f_{vu} = 0 \\
& \sum_{(u,v)\in E} f_{uv} \leq k \\
& \forall (u, v) \in E : 0 \leq f_{uv} \leq 1
\end{aligned}$$

Note that this is the same formulation as a flow problem of sending one unit of flow from s to t with the added condition that we can use at most k edges.

b) Show that the (optimal) value of the obtained LP is upper bounded by the shortest $s - t$ path with at most k edges.

Solution: Observe that if we change the above LP to have the unknowns f_{uv} be in $\{0, 1\}$ instead of $0 \leq f_{uv} \leq 1$, we have that the problem will return the shortest $s - t$ path with at most k edges. Once we relax the problem with the constraint that $0 \leq f_{uv} \leq 1$, the optimal value of the LP can only drop because the set of feasible solutions to the original integer linear program is a subset of the feasible solutions of the relaxed linear program we formulated above. Thus, the optimal value of the above LP is upper bounded by the shortest $s - t$ path with at most k edges.

c) Show that if the value of your LP is v, then there exists a shortest path of k edges and of weight v. *Hint:* use flow decomposition theorem. Can any path (with non-zero flow) have length more than k ?

Solution: Suppose the value of the LP is v. Since we can view the LP as a flow problem, by the flow decomposition theorem, we can also decompose the output flow into a series of feasible flows with non-zero flow and corresponding paths. Denote these paths as $P_1, \ldots, P_j$ with flow values $q_1, \ldots, q_j \in [0, 1]$. We will then show that it must be that one of these P_i has the same weight as the shortest path with k edges.

First, we observe that no P_i can have length more than k. Suppose it did. Then, we could simply reduce the overall flow by finding a path of length equal to k, contradicting the fact that the final flow is an optimal solution. To prove the claim that one of the P_i must be the same weight as the shortest path with k edges, we also prove by contradiction. Suppose that no P_i has the same weight as the shortest path with k edges, so all P_i have higher cost. Then, we can always move flow from some P_i to the shortest path with k edges, decreasing the overall flow and contradicting our assumption that the final flow is optimal. Thus, we have shown that there always exists a shortest path of k edges and of weight v if the value of the LP is v.

Problem 3

Show that for any decision problem in NP, there is an algorithm that can solve it that runs in time $2^{O(n^k)}$, for some constant $k > 0$.

Solution:

Since the problem is in NP, there is some polynomial-time verifier A for it such that for any input x of length n, there is a certificate y of length polynomial in n such that $A(x, y) =$ true. In particular, since y must be polynomial in n, we know that there exists some constant k such that we only have to check certificates of size $O(n^k)$.

The certificate string is a binary string. Since it has length $O(n^k)$ there are at most $2^{O(n^k)}$ possible certificates. Thus, our algorithm is as follows: given any input x, run $A(x, y)$ for all possible certificates y. If $A(x, y) =$ true for any certificate y, then input x has a solution; else, if $A(x, y) =$ false for *all* certificates y, input x has no solution.

Given input x we run $A(x, y)$ for $2^{O(n^k)}$ certificates y, and each run takes polynomial time (since A must be a poly-time verification algorithm), so the total run-time is $2^{O(n^k)} \cdot \text{poly}(n)$. It is not hard to see that the poly(n) factor gets swallowed up in the $O(n^k)$ in the exponent, leading to an overall run-time of $2^{O(n^k)}$.

(You only had to mention that the poly(n) term gets swalloed up the $O(n^k)$ in the exponent; you didn't have to go into details. But I'll do it our here for those of you who want to see the formal proof. poly(n) means $d \cdot n^c$ for some constants d, c. But $d \cdot n^c < 2^{cd \log(n)}$ and $cd \log(n) = O(\log(n)) = O(n^k)$, so

$$2^{O(n^k)} \cdot d \cdot n^c < 2^{O(n^k)} 2^{cd \log(n)} = 2^{O(n^k) + cd \log(n)} = 2^{O(n^k)}$$

Problem 4 (optional for sections 002 and H02)

Given an integer $m \times n$ matrix A and an integer m-vector b, the *0-1 integer-programming problem* asks whether there exists an integer n-vector x with elements in the set $\{0, 1\}$ such that $Ax \leq b$. Prove that 0-1 integer programming is NP-complete. (Hint: Reduce from 3-SAT).

Solution: First we show that the 0-1 integer-programming problem is in NP. This is obvious, because if we find an array x, we can certainly do matrix-vector multiplication and vector comparison in polynomial time.

Then we argue that the 0-1 integer-programming problem is NP-hard by showing that every 3-CNF-SAT problem can be reduced to an 0-1 integer-programming problem in polynomial time.

Suppose a 3-CNF-SAT has l clauses and k variables, $x_1, x_2, \ldots, x_k$. We define new variables $\bar{x}_i = \neg x_i$, for $i = 1, \ldots, k$. Equivalently we have $x_i + \bar{x}_i = 1$, which means

$$\begin{aligned} x_i + \bar{x}_i &\leq 1 \\ -x_i - \bar{x}_i &\leq -1. \end{aligned}$$

Then for a clause of 3-CNF, we have following table (by DeMorgan):

satisfiable formulae	equivalent inequality
$x_i \vee x_j \vee x_k$	$\bar{x}_i + \bar{x}_j + \bar{x}_k \leq 2$
$x_i \vee x_j \vee \neg x_k$	$\bar{x}_i + \bar{x}_j + x_k \leq 2$
$x_i \vee \neg x_j \vee x_k$	$\bar{x}_i + x_j + \bar{x}_k \leq 2$
$x_i \vee \neg x_j \vee \neg x_k$	$\bar{x}_i + x_j + x_k \leq 2$
$\neg x_i \vee x_j \vee x_k$	$x_i + \bar{x}_j + \bar{x}_k \leq 2$
$\neg x_i \vee x_j \vee \neg x_k$	$x_i + \bar{x}_j + x_k \leq 2$
$\neg x_i \vee \neg x_j \vee x_k$	$x_i + x_j + \bar{x}_k \leq 2$
$\neg x_i \vee \neg x_j \vee \neg x_k$	$x_i + x_j + x_k \leq 2$

(Note that there are several different, and equivalent ways to formulate these inequalities, depending on whether you use x_i or $\bar{x}_i$. For each clause of 3-CNF, we have an equivalent inequality, and for each variable in 3-CNF, we have two equivalent inequality, thus we have $l + 2k$ equivalent inqualities and $2k$ variables within. We have constructed an 0-1 integer-programming problem, in which $m = l + 2k$ and $n = 2k$, and $x = [x_1, \bar{x}_1, \ldots, x_k, \bar{x}_k]$. A and b are constructed from 3-CNFs by above inequality. The possible values of entries of A are -1, 1 and 0, and possible values of elements of b are 1, -1 and 2. Apparently if the 3-CNF is satisfied, there equivalent inequality are satisfied, and vice versa, if $Ax \leq b$ is satisfied, each clause of 3-CNF must be satisfied, and vice versa. This reduction takes polynomial time, because for each clause it takes constant time to fill up non-zero element in each line of A and b. Then we conclude that the 0-1 integer-programming problem is NP-complete.