

CS599 (Spring 2007) - Midterm Solutions

Problem 1

We use rounding of an appropriate LP. The integer program has variables s_i, t_i , which are 1 if we pick S_i (respectively T_i), and 0 otherwise. In addition, in order to express the objective function, we have one variable y_e for each element $e \in U$, which is 1 if element e is included in one or more picked set, and 0 otherwise. We need to have a constraint that exactly one of S_i and T_i is picked for each i (hence, $s_i + t_i = 1$ for all i). In addition, we need to express that if S_i is picked, then so must be all elements in S_i . In other words, if $s_i = 1$, then $y_e = 1$ for all $e \in S_i$. Thus, we have that $y_e \geq s_i$ for all $e \in S_i$. Of course, we have a similar constraint for T_i . The full integer program is then

$$\begin{aligned} & \text{Minimize} && \sum_e y_e \\ & \text{subject to} && s_i + t_i = 1 && \text{for all } i \\ & && y_e \geq s_i && \text{for all } i \text{ and } e \in S_i \\ & && y_e \geq t_i && \text{for all } i \text{ and } e \in T_i \\ & && s_i, t_i, y_e \in \{0, 1\} \end{aligned}$$

As usual, the LP relaxation is obtained by replacing the last constraint with $s_i, t_i, y_e \geq 0$. This LP has polynomially many variables and constraints, and can thus be solved in polynomial time. The rounding is then very simple: for each i , include S_i if and only if $s_i \geq \frac{1}{2}$, and include T_i otherwise.

Clearly, this gives a feasible solution (it includes one set for each i). The elements included by the algorithm are exactly those e with $e \in S_i$ and $s_i \geq \frac{1}{2}$ (for some i), or $e \in T_i$ and $t_i > \frac{1}{2}$ (for some i). In both cases, the second and third constraint of the LP imply that $y_e \geq \frac{1}{2}$. Hence, all elements included had $y_e \geq \frac{1}{2}$, whereas $y'_e = 1$ in the integral solution. Thus, the cost of the solution at most doubles (for all non-included elements, it is possible that $y'_e < y_e$, but certainly not $y'_e = 0 > y_e$ by non-negativity). So the algorithm is a 2-approximation.

Problem 2

We will first derive a pseudo-polynomial time algorithm, and then from it an FPTAS. Let $\hat{d} = \max_e d_e$ denote the largest delay of any edge. For the dynamic program, we will express a recurrence for the smallest cost for reaching any node v with total delay at most d . Let $a(v, d)$ be the smallest cost with which node v can be reached at delay d or less. The base case is that $a(s, d) = 0$ for all d . For nodes other than s , we have that $a(v, d) = \min_{e=(u,v), d_e \leq d} (a(u, d - d_e) + c_e)$. So the minimum is taken over all edges e into v , and we define that the minimum is ∞ if there are no edges e into v with $d_e \leq d$. Finally, we are looking for the smallest d such that $a(t, d) \leq C$ (if any; otherwise, there is no path with corresponding cost).

This DP is correct because the optimal path of delay d or less into v must have some last hop, and we are checking for that hop. Notice that the recurrence is not cyclic because the graph was a DAG. We can process the nodes layer by layer. Also, once we have computed the table of $a(t, d)$, we can backtrack through the table to read off the solution (or compute it along the way). The running time is $O(n^3 \hat{d})$, because the maximum possible value of d in the table is $n\hat{d}$, so the table has $O(n^2 \hat{d})$ entries, each taking time $O(n)$ to compute.

To turn the DP into an FPTAS, we will first make an additional assumption: that $\hat{d} \leq d(\text{OPT})$, where $d(\text{OPT})$ is the total delay of the optimum solution. That is, the maximum delay on any edge is at most that of the entire optimum solution. We later show how to get around that assumption. For now, we apply the standard technique from KNAPSACK. Given an ϵ , we replace each delay d_e by $d'_e := \lfloor \frac{n}{\epsilon \hat{d}} \cdot d_e \rfloor$. All delays are now integers between 0 and n/ϵ , so we can solve the dynamic program in time $O(n^4/\epsilon)$, which is polynomial in n and ϵ .

The dynamic program finds the optimum solution with respect to the d'_e . When we return this solution, the true delay of each edge e is at most $\frac{\hat{d}\epsilon}{n}(d'_e + 1)$, so the total delay of the path P returned by the algorithm is at most $\frac{\hat{d}\epsilon}{n} \sum_{e \in P} d'_e + \hat{d}\epsilon$ (because P has at most n edges). Each of the edges of the optimum path OPT

has real delay at least $\frac{\hat{d}\epsilon}{n}d'_e$, so the total delay is at least $\frac{\hat{d}\epsilon}{n}\sum_{e\in\text{OPT}}d'_e$. OPT was no better than P with respect to the d'_e , so we obtain that $d(P)\leq d(\text{OPT})+\hat{d}\epsilon$. By the assumption that $\hat{d}\leq d(\text{OPT})$, this is bounded by $(1+\epsilon)d(\text{OPT})$, giving us an FPTAS.

Finally, to ensure the assumption that $\hat{d}\leq d(\text{OPT})$, we can reason as follows. In the true optimum solution OPT, there is some edge \hat{e} which has largest delay among all $e\in\text{OPT}$. If we removed from G completely all edges e with $d_e>d_{\hat{e}}$, then it would certainly be true in the remaining graph that $\hat{d}\leq d_{\hat{e}}\leq d(\text{OPT})$, and the optimum solution would not change. Since we don't know which edge actually is \hat{e} , we can simply try all of them; for each solution P_e that we find after removing all edges with delay larger than d_e , we compute how good the solution really is (with respect to the true delays d , not d'), and keep only the best one. That way, we run the dynamic program at most $O(n^2)$ times, and at the end are sure to have tried the correct \hat{e} . The total running time is then just $O(n^6/\epsilon)$.

Problem 3

The algorithm is quite simple: run the greedy algorithm always picking next the set S minimizing $c_S/|S\cap R|$ (where R is the set of remaining elements), but stop as soon as 90% of all elements are covered. This algorithm clearly solves the problem, and we will show that it is roughly a 3.4-approximation.

The key intuition is that in the greedy algorithm, the sets added at the end are really the expensive ones. More precisely, let p_i be the price of the i^{th} element the algorithm adds (ties broken arbitrarily). That is, $p_i=c_{S_i}/|S_i\cap R_i|$, where S_i is the set used to cover element i for the first time, and R_i is the set of elements remaining at the moment right before i is covered. Then, the total cost of the algorithm is exactly $\sum_i p_i$. If an element is not covered at all, we define its price to be 0. Notice that the number of elements actually covered may be anywhere from 90%–100% of all elements.

We proved in class the Lemma 2.3 from the textbook, which shows that $p_i\leq\text{OPT}/(n-i+1)$ for all i . This lemma continues to hold for all i , as we didn't change anything about the algorithm except its termination condition. Furthermore, for $i\geq 0.9n$, we know that the only way element i is covered is because we picked a larger set in the last iteration (covering more elements than necessary). Still, at the time the element i is covered, we know that at least $n/10$ elements were uncovered. Hence, the cost effectiveness of the optimum solution was no worse than $\text{OPT}/0.1n$, which means that the set picked by the greedy algorithm also had cost effectiveness no worse than $\text{OPT}/0.1n$. Thus, for $i\geq 0.9n$, we have proved a bound of $p_i\leq\text{OPT}/0.1n$. (This holds trivially for uncovered elements i , as $p_i=0$ for those.)

Thus, the total cost of the algorithm is bounded from above by

$$\begin{aligned}\sum_{i=1}^n p_i &\leq \sum_{i=1}^{0.9n} p_i + \sum_{i=0.9n}^n p_i \\ &\leq \sum_{i=1}^{0.9n} \text{OPT}/(n-i+1) + \sum_{i=0.9n}^n \text{OPT}/0.1n \\ &= \text{OPT} \cdot \sum_{i=0.1n}^n 1/i + \text{OPT} \\ &\leq \text{OPT} \cdot (\ln n - \ln(0.1n)) + \text{OPT} \\ &= \text{OPT} \cdot (1 + \ln(10)) \\ &\leq 3.4 \cdot \text{OPT}.\end{aligned}$$

Hence, the algorithm is a 3.4 approximation.

Problem 4

The primal LP (or, to be precise, IP) has one variable x_e for each edge e , saying whether the edge is included in the path (value 1) or not (value 0). Then, for each cut separating s and t , there must be at least one

edge crossing it, and subject to that constraint, we want to minimize the total cost. Thus, the primal LP relaxation is

$$\begin{aligned} & \text{Minimize} && \sum_e c_e x_e \\ & \text{subject to} && \sum_{e \in (S, \bar{S})} x_e \geq 1 && \text{for all } s\text{-}t \text{ cuts } (S, \bar{S}) \\ & && x_e \geq 0 && \text{for all } e. \end{aligned}$$

The dual has one variable y_S for each s - t cut (S, \bar{S}) , and is as follows:

$$\begin{aligned} & \text{Maximize} && \sum_S y_S \\ & \text{subject to} && \sum_{S: e \in (S, \bar{S})} y_S \leq c_e && \text{for all edges } e \\ & && y_S \geq 0 && \text{for all } S. \end{aligned}$$

Notice that the primal has exponentially many constraints, but that doesn't bother us for a primal-dual algorithm. The dual has exponentially many variables, but our algorithm will only use polynomially many.

The algorithm follows the usual pattern: raise one dual variable until the constraint of one or more edge becomes tight. Include all tight edges (temporarily) in the primal solution. Then raise more dual variables and repeat until we have an s - t path in the solution. Afterwards, run reverse delete to get rid of all edges that are not needed.

The order in which the dual variables are raised is as follows: first, we raise the dual variable y_S for $S = \{s\}$. When one or more edge out of s becomes tight, and all those edges are added to the primal solution, include all endpoints of those edges in the new set S (in addition to s), and raise y_S . Continue including the endpoints of tight edges to obtain the next set. Stop as soon as t is included in S . (It turns out that this is exactly what Dijkstra's algorithm does to build a shortest path tree out of s . The reverse delete step then removes all the branches of the tree that did not lead to a shortest path to t .)

The reverse delete step, as usual, looks at all edges in the reverse order from the one in which they were added. If an edge e is not needed (i.e., there still is an s - t path even if e is removed), then e is greedily removed from the primal solution.

When the primal-dual algorithm terminates, there is an s - t path in the solution. We can prove this by induction: whenever a new node v is added to S , there is a path from s to v using just edges of the primal solution. In the base case $S = \{s\}$, this clearly holds. Later, we only add endpoints of edges out of S , and by hypothesis, each node in S at that time can be reached from s . The reverse delete step only removes edges if a solution still exists afterwards, so we still have a feasible solution at the end.

To prove optimality, we use the primal and dual complementary slackness conditions. For the primal condition, we know that the algorithm only ever includes edges when their constraints are tight; hence, for each e with $x_e = 1$, we have that $\sum_{S: e \in (S, \bar{S})} y_S = c_e$.

For dual complementary slackness, let (S, \bar{S}) be an s - t cut with $y_S > 0$. In particular, the algorithm must at some point have raised y_S . Assume for contradiction that the primal solution contains multiple edges crossing (S, \bar{S}) . Among these edges, let $e = (u, v)$ be the first one that was added, and $e' = (u', v')$ another edge not deleted by the reverse delete step. At the time e is considered for removal, all nodes in S , in particular u and v , are still reachable from s , because all edges inside S were added before the first edge across (S, \bar{S}) . Thus, the primal solution also still contains a path from s to v not using e' , and a path from s to v' not using e . But at least one of v, v' must be able to reach t without going through the other; hence, the corresponding other one among e or e' would have been deleted in the reverse delete step.

Thus, each cut with $y_S > 0$ is crossed by only one edge, giving us dual complementary slackness, and thus that the returned solution is optimal. The algorithm runs in polynomial time because it only raises at most n dual variables, and for each, we can determine in time $O(m)$ when the first edge becomes tight (really, we can do it faster; see Dijkstra). Reverse delete clearly takes only $O(n^2)$ time, even if implemented very naively, with repeated BFS.