Massachusetts Institute of Technology — Handout 20
6.854J/18.415J: Advanced Algorithms — Wednesday, November 9, 2005
David Karger

# Problem Set 9 Solutions

**Problem 1.**

**(a)** Let $S$ be an independent set in $G$. If in the product graph we choose from each $G_v$, where $v \in S$, a copy of set $S$, we will get a set $S'$ of size $|S|^2$. $S'$ is an independent set, since there is no edge between copies $G_v$ for $v \in S$, and inside a copy there is no edge between vertices $v$ for $v \in S$.

**(b)** Let $S$ be the independent set of size $s$ in the product graph of $G$. If elements of $S$ belong to at least $\sqrt{s}$ copies $G_v$ of the original graph, we are done, since these copies correspond to vertices that are independent in $G$. Otherwise, by the pigeon hole principle, there must be at least one copy containing $\sqrt{s}$ vertices, and they constitute an independent set of size at least $\sqrt{s}$.

Note that the proof that we have just presented is constructive, and shows how to get an independent set of size at least $\sqrt{s}$, knowing an independent set of size $s$ in the product graph in polynomial time.

**(c)** Suppose that there is an $\alpha$-approximation algorithm ($\alpha > 1$). For a given $\epsilon > 0$ and for some large enough integer $k$, it holds that

$$1 + \epsilon \geq \sqrt[2^k]{\alpha}.$$

Let OPT be the size of the maximum independent set in our graph $G$. We take $G = G_0$, and construct a product graph $G_1$ of it, then the product graph $G_2$ of $G_1$, and so on, till we achieve the product graph $G_k$ of $n^{2^k}$ vertices. $G_k$ has an independent set of size $\mathrm{OPT}^{2^k}$. We find an $\alpha$-approximation of an independent set in $G_k$, and using the algorithm from part $(b)$ we find an independent set in $G$ of size at least

$$\frac{\mathrm{OPT}}{\sqrt[2^k]{\alpha}}.$$

The algorithm obviously runs in polynomial time for fixed $\epsilon$.

**Problem 2.**

**(a)** We will show that the cost of the minimum spanning tree in $G'$ is at most two times greater than the cost of the optimum steiner tree $S$ in $G$.

We can run the DFS algorithm from an arbitrary vertex $v$ in $S$, and order terminals $T$ according to a time of the first visit. Let $t_1$, $t_2$, ..., $t_k$ be these vertices in the just specified order. Let us consider a traversal starting in $t_1$, and ending in $t_k$, visiting consecutively $t_2$, $t_3$, and so on. From $t_i$ to $t_{i+1}$ we go along some shortest path from $t_i$ to $t_{i+1}$. It is clear that this traversal corresponds to some spanning tree in $G'$, namely, we take edges $(t_i, t_{i+1})$ for $1 \leq i < k$, so the cost of the optimal spanning tree in $G'$ is not greater than the cost of our traversal.

Our claim is that the cost of the traversal is at most two times greater than the cost of $S$. This is true, since if we went from a vertex to the next along edges that belong to $S$, as we did in the DFS, we would go along each edge at most twice.

**(b)** We will use the knowledge from part (a) to construct a 2-approximation algorithm. It will create a tree connecting all the terminals in the fashion of Prim's algorithm. We start from an arbitrary terminal, find the shortest path from it to another terminal, and connect them by this path. Then, until all terminals are connected, we in turn find a terminal that is closest to the already created tree, and connect it by some shortest path. Obviously, we will eventually get a tree, and since its cost will not be greater than the cost of the optimal spanning tree in $G'$, we get in polynomial time a 2-approximation for the Steiner tree problem.

**Problem 3.**

**(a)** Note first that the number of all different subsets of the set of items is polynomial. Let $n$ be the number of all items. The number of different subsets can be bounded by $C = (n+1)^k$. This also means that there are at most $C$ different configuration of bins.

We will compute sets $S_i$. $S_i$ is the set of all possible sets of items left when $i$ bins has been already packed. Obviously, $S_0$ contains only one subset, the set of all items. $S_{i+1}$ can be computed from $S_i$ in polynomial time, for each item in $S_i$ we try all possible packings, and each time we add the set of items that are left to $S_{i+1}$. We stop when in a new computed $S_j$ there is an empty subset. This implies that the optimum packing of items uses $j$ bins. The optimal number of bins can be upper bounded by $n$, since each item can be put into a separate bin. We can compute sets $S_i$ and reconstruct the optimum packing in polynomial time.

**(b)** Starting over the first bin, we will consider items we want to add one by one. We check whether it is possible to add an item to the current bin. If it is, then simply do it. Otherwise, we move to the next bin (possibly adding a new bin), try to add an item there, and so forth, until we eventually put it into some bin. We do the same with any other item, starting always over the bin to which the last item was inserted. The algorithm runs in time linear in the number of items and bins.

Suppose first that $\epsilon \in (0, 1/2]$. For such $\epsilon$ it can be simply proven that

$$\frac{1}{1-\epsilon} \leq 1 + 2\epsilon.$$

When we cannot add an item to a current bin, it follows by the upper bound on size of items that it must contain items of the sum of sizes greater than $1 - \epsilon$. Since the sum of sizes of all items is bounded by $B^*$, we are able to fit all items of size at most $\epsilon$ into the first

$$\left\lceil \frac{B^*}{1-\epsilon} \right\rceil < \lceil (1+2\epsilon)B^* \rceil \leq 1 + (1+2\epsilon)B^*$$

bins, and therefore we can achieve a packing of at most $\max\{B, 1 + (1+2\epsilon)B^*\}$ bins.

When $\epsilon > 1/2$, it suffices if we can fit all items into $2B^*$ bins, since

$$1 + (1+2\epsilon)B^* > 2B^*.$$

We shall show that the average fill ratio of bins (possibly with exclusion of the last one) is at least $1/2$, what will in turn bound the number of bins by required $2B^*$. The first $B$ bins contain elements greater than $\epsilon$, so even if we do not manage to add something to them, they are filled enough for our purpose. Consider a pair of consecutive bins that we add over the initial $B$ bins. At some moment we cannot put an item into the first bin, and therefore we put it into a new bin, that is into our second bin. It implies that in both bins there are items of the sum of sizes greater than 1, that is at least $1/2$ on average in both of them. This proves our claim on the average fill ratio.

**(c)** In bin packing, even small enlargement of sizes of items may double the required number of bins we need (if almost all $a_i = 1/2$). It was not the case for $P||C_{\max}$ because loads of machines were not bounded, and we were minimizing a maximum load, not the number of machines.

**(d)** We put aside first $n/k$ items—they can be placed each into its own bin. Notice that after the grouping procedure the $(n/k) + i$-th item has size not greater than the $i$-th item before grouping, and therefore all items that do not belong to the first group can be put into the same number of bins as the original items, if we take the $(n/k) + i$-th item after grouping instead of the $i$-th item before the grouping. Eventually, the optimal number of bins increases by at most $n/k$.

**(e)** For a given $\epsilon$ we set $k = 2/\epsilon^2$. Let $m$ be the number of items of size greater than $\epsilon/2$. We know that

$$m\frac{\epsilon}{2} \leq B^*.$$

First, we apply the grouping from part (d), and we know by part (a) that using the algorithm from part (a), we can find the optimum packing for modified sizes of size at most the ceiling of

$$B^* + \frac{m}{k} = B^* + \frac{m\epsilon^2}{2} \leq B^* + \epsilon B^* = (1+\epsilon)B^*,$$

i.e. of size

$$B^*(1+\epsilon) + 1.$$

By part (b) we know that we can add items of size not exceeding $\epsilon/2$ in linear time, fitting into $B^*(1+\epsilon)+1$ bins. The algorithm works in polynomial time.

**Problem 4.**

**(a)** An integer variable $x_{jt}$, for $j = 1 \ldots n$, where $n$ is the number of jobs, and $t = 1 \ldots \sum_j p_j$, is an indicator that job $j$ completed at time $t$. Obviously,

$$\forall j\ \forall t \qquad 0 \leq x_{jt} \leq 1.$$

We enforce that every job completes:

$$\forall j \qquad \sum_t x_{jt} = 1,$$

that a job is not processed before its predecessor:

$$\forall j\ \forall k \in A(j)\ \forall t \qquad \sum_{i=1}^{t-p_k} x_{ki} \geq \sum_{i=1}^{t} x_{ji},$$

and that the total processing time of jobs completed before time $t$ is at most $t$:

$$\forall t \qquad \sum_{i=1}^{t} \sum_j x_{ji} \leq t.$$

We only need to specify a goal function. It is

$$\min \sum_t \sum_j w_j t x_{jt}.$$

**(b)** For a job $j$ and its halfway point $h_j$

$$\overline{C}_j = \sum_t t x_{jt} \geq \sum_{t \geq h_j} t x_{jt} \geq \sum_{t \geq h_j} h_j x_{jt} = h_j \sum_{t \geq h_j} x_{jt} \geq \frac{1}{2} h_j.$$

Voilà!

**(c)** The set of constraints that worked for the integral linear program, and expressed that a job $j$ cannot be processed before its predecessor $k$, now says that a half of $j$ must be processed after a half of $k$. This implies that no job runs before its predecessor.

**(d)** Note that at the halfway point of job $j$ at least half of each of jobs proceeding $j$ in the order has been completed. This implies that a half of the sum of their (including $j$) processing times is not greater than $h_j$, and therefore the completion time of $j$ is at most $2h_j$. By part (b) we know in turn that their completion time is not greater than $4\overline{C}_j$.

**(e)** Solving the linear program, we minimize function

$$\sum_t \sum_j w_j t x_{jt} = \sum_j w_j \sum_t t x_{jt} = \sum_j w_j \overline{C}_j.$$

The minimum we achieve is not worse than the minimum for the integer linear program, and because our completion times are not worse than four times average completion times in the optimal solution, we achieve a constant factor-approximation for $1 \,|\, prec \,|\, \sum C_j$.