

Problem Set 7 Solutions

Fri, October 30, 2009

Problem 1. (a) The algorithm repeats the following step k times, or until $S = \emptyset$:

Arbitrarily pick a point $p \in S$, and let $C = \{p' \mid d(p, p') \leq d\}$. Set aside C as a cluster, and set $S = S \setminus C$.

We claim that this algorithm is a 2-approximation. By the triangle inequality, the diameter of every cluster is at most $2d$. Furthermore, we argue that at the end of the algorithm, every point is in exactly one cluster. It is clear that each point is in at most one cluster. Now, assume by way of contradiction that there is some point q that was not assigned a cluster. Then it must be at distance greater than d from every point p around which we chose a cluster. However, each of these points is at distance greater than d , which implies any k clustering will have diameter greater than d , contradicting the optimality of d .

(b) Let the algorithm above be A , and the proposed algorithm B . In each iteration, algorithm B selects a point as a center. Note that algorithm A does not specify which point to choose in each iteration; it picks an arbitrary point among the points not already clustered.

We show that in each iteration, algorithm A can either choose the exact point algorithm B chooses as a center, or if it cannot, algorithm B is already a 2-approximation.

If algorithm A chooses the exact same set of points as algorithm B , and algorithm A is a 2-approximation, it follows that algorithm B is a 2-approximation. The clustering in algorithm A shows that each point is at distance at most d from each point chosen; as a result, assigning each point to its closest center will result in a 2-approximation. We now prove the necessary lemma.

Lemma 0.1 *Let p be the center algorithm B chooses in a given iteration i . Either algorithm A can choose p as a center of a cluster in iteration i , or algorithm B has already achieved a 2-approximation.*

Proof. Consider a given iteration. If algorithm A cannot choose p , this implies that p is at distance at most d from each of the centers of the clusters already created. But if algorithm B chooses p , it implies that every point is at distance at most d from the centers already chosen, which implies that assigning each point to its closest cluster gives a 2-approximation. ■

Problem 2. (a) Consider any feasible subset S of jobs, i.e. a set of jobs that together can be completed by their due dates. Then we can schedule this set of jobs in terms of increasing deadline. We argue this by a swapping argument: consider a schedule π of S in which each job is completed before its due date.

Apply the following until it is no longer possible: if there are two jobs i, j such that i is scheduled before j , but $d_i > d_j$, swap them. Now, job j obviously is scheduled before its due date. Job i is also scheduled before its due date, because it completes at time before d_j (otherwise, job j was not completed without penalty in π), and $d_j < d_i$.

When we can no longer apply the above, the sequence of jobs is sorted by increasing due dates.

(b) Recall we have a set of jobs $\{1 \dots n\}$ with processing times $\{p_1 \dots p_n\}$, deadlines $\{d_1 \dots d_n\}$ and weights $\{w_1 \dots w_n\}$. We wish to find $J \subseteq \{1 \dots n\}$, the feasible subset of maximum weight with the fastest completion time. By part (a), we can schedule a feasible subset in the order of increasing deadlines, so we sort the jobs accordingly so $d_i \leq d_{i+1}$ for all i .

We set up a dynamic program as follows. Our table F will have n rows and $W = \sum_j w_j$ columns. We define

$$F(i, w) = \min\{p(J) \mid J \subseteq \{1 \dots i\} \text{ is feasible, } w(J) = w\}$$

where $p(J) = \sum_{j \in J} p_j$ and $w(J) = \sum_{j \in J} w_j$.

For our base case, we set $F(i, 0) = 0$, for all i . This means that the empty subset (which has weight 0) can be completed in no time. We additionally set $F(0, w) = \infty$ for all w , i.e. no empty subset of jobs has weight w . The remaining entries will be entered using the following rule:

$$F(i, w) = \begin{cases} \min\{F(i-1, w), p_i + F(i-1, w - w_i)\} & \text{if } F(i-1, w - w_i) + p_i \leq d_i \\ F(i-1, w) & \text{otherwise} \end{cases}$$

The running time of this algorithm is $O(nW)$, which is polynomially bounded in n under our assumption.

(c) An unfortunate mistake was assuming that a $(1 - \epsilon)$ approximation scheme for the problem of finding the (fastest-completing) maximum-weight feasible subset was *equivalent* to a $(1 + \epsilon)$ approximation scheme for finding the schedule with minimum lateness penalty.

Let the total weight be W , and let W_A be the weight of the maximum weight feasible subset. Let W_B be the weight (penalty) of the schedule that minimizes lateness penalty. It is true that:

$$W = W_A + W_B$$

However, suppose we wished to obtain a $(1 + \epsilon)$ approximation scheme for the minimization problem, and we have a $(1 - \delta)$ approximation scheme for the maximization problem. If we use the $(1 - \delta)$ approximation scheme naively, we could say that we achieve a schedule that has cost at most:

$$W - (1 - \delta)W_A$$

We want this to be at most $(1 + \epsilon)W_B$. Let us see what this requires:

$$\begin{aligned} W - (1 - \delta)W_A &\leq (1 + \epsilon)W_B \\ &\iff \\ \delta &\leq \frac{(1 + \epsilon)W_B - W + W_A}{W_A} \\ &\iff \\ \delta &\leq \frac{\epsilon W_B}{W_A} \end{aligned}$$

But $\frac{W_B}{W_A}$ could be arbitrarily small; in particular, for the FPTAS we have for maximizing W_A , we would no longer necessarily be strongly polynomial (since $\frac{1}{\epsilon}$ now involves the magnitude of the weights).

The key to this problem is using part (b) in an exact way.

Recall that we are trying to minimize lateness, i.e. $\sum_j w_j U_j$, and we desire a $(1 + \epsilon)$ approximation scheme. Let the value of the optimum schedule be OPT , and let us assume for the moment that this value is known to us. We first multiply each weight w_j by $\frac{n}{\epsilon \text{OPT}}$ to get a weight w'_j . In the instance of the problem with the scaled weights, $\text{OPT}' = \frac{n}{\epsilon}$.

Note that a schedule with value $(1 + \epsilon)\text{OPT}'$ for the instance with scaled weights w'_j is a schedule with value $(1 + \epsilon)\text{OPT}$ for the regular weights w_j . Our goal, then, will be to find a schedule of cost $(1 + \epsilon)\text{OPT}' = n + \frac{n}{\epsilon}$.

Consider the weights w' ; note that they are not integral. We round up each one to the nearest integer to obtain weights w^* , which results in $\text{OPT}^* \leq \text{OPT}' + n = \frac{n}{\epsilon} + n$. (Alternatively, if we round down to the nearest integer, $\text{OPT}' \leq \frac{n}{\epsilon}$, and undoing the rounding will make the solution worse by at most n , so we still have the same result). Now, as long as the weights are of polynomial size in $\frac{n}{\epsilon}$, we can compute the optimum solution in part (b), which is of value at most $\frac{n}{\epsilon} + n$, which is at most $(1 + \epsilon)\text{OPT}'$, which is what was desired. This isn't true if some weight of a job is much greater than OPT , but if that's the case, we know it *must* be scheduled before its due date; otherwise, the schedule would have a lateness greater than that of OPT . So we schedule all of these big weights first, and then do the dynamic program on the remaining weights. The dynamic program runs in time $O(\frac{n^3}{\epsilon})$.

This algorithm assumes we know the value of OPT . We can binary search over $[0, \sum_j w_j]$ to obtain a good guess of OPT . In each binary search iteration, we

will run the above algorithm. Note that if we overguess OPT , our solution will be greater than $(1 + \epsilon)\text{OPT}'$, and so it will be greater than $(1 + \epsilon)$ of our guess. If we underguess OPT , our solution will be less than $(1 + \epsilon)$ of our guess. We will binary search and find the greatest such guess such that our solution is $(1 + \epsilon)$ of our guess.

Problem 3. (a) We can essentially use the dynamic program used for $P\|C_{\max}$ (as shown in class). Since there are only k different job sizes, there are $O(n^k)$ bin profiles for picking how many of the $O(n)$ items of each size go into the bin. Each step in the dynamic program calculates what sets of items can fit in i bins based on the sets of sets found for $i - 1$ bins by trying to add a bin profile to each existing set of $i - 1$ bins. This takes $O(n^{2k})$ per step, so the minimum number of bins possible can be found in time $O(n^{2k+1})$.

(b) Simply go through the bins one by one, and put items into them until they have less than ϵ space left. If we did not need extra bins, we are done. Otherwise, notice that at the end of this procedure, all the bins except possibly the last one have at least $1 - \epsilon$ of “material” in them. If we have used $x + 1$ bins, there is at least $x(1 - \epsilon)$ “material” in total, so $B^* \geq x(1 - \epsilon)$. Therefore, $x + 1 \leq 1 + B^*/(1 - \epsilon)$. For $\epsilon \leq 1/2$, $1/(1 - \epsilon) \leq 1 + 2\epsilon$, so we get $x + 1 \leq 1 + (1 + 2\epsilon)B^*$.

For $\epsilon > 1/2$, then we know that all the items with size greater than ϵ had to be placed in their own bins, and each of these bins is at least half-full. In any of the new bins used in the greedy algorithm, any two consecutive bins must contain at least 1 unit size of material between them or else they could have both fit into one bin. Thus all of the new bins are at least half-full on average as well. Thus we have $B' \leq 2 \sum_i a_i \leq 2B^* < 1 + (1 + 2\epsilon)B^*$ for $\epsilon > 1/2$.

Thus the greedy algorithm uses either B bins if no extra bins are necessary and at most $1 + (1 + 2\epsilon)B^*$ bins otherwise.

(c) Modifying item sizes by rounding does not work for this problem because even a small absolute modification of item sizes might affect the number of bins by a factor of 2 (if each item has size about $1/2$).

(d) Let q_i be the size of the largest item in S_i . Note that every item in S_{i-1} is at least as big as q_i . Take an optimal packing and pack the items of increased size as follows: replace items in S_{i-1} in the packing by items in S_i (each item size is now q_i). Put the items in S_1 each into its own separate bin. This uses at most n/k extra bins.

(e) Let ϵ be given and let $k = 2/\epsilon^2$. The grouping procedure above in part (c) and the DP of part (a) packs all items of size at least $\epsilon/2$ into at most $B^* + m\epsilon^2/2$ bins where m is the number of items of size at least $\epsilon/2$. But one can't pack these m items into any fewer than $m\epsilon/2$ bins, i.e. $m\epsilon/2 \leq B^*$. Therefore, $m\epsilon^2/2 \leq \epsilon B^*$. Thus, we have packed the largest items into $B^*(1 + \epsilon)$ bins. Now add the items

smaller than $\epsilon/2$ using the method in part (b). After the process, we have used at most $\max\{(1 + \epsilon)B^*, 1 + (1 + 2(\epsilon/2))B^*\}$ bins, so we have the desired bound of at most $1 + (1 + \epsilon)B^*$ bins.

- Problem 4.** (a) If we break each vertex into two, an “entry node” and an “exit node”, the problem of finding a cycle cover is simply a bipartite matching between entry nodes and exit nodes. To find the minimum cycle cover, simply solve the min-cost maximum matching algorithm by using the min-cost max-flow algorithm as shown in class.
- (b) Each cycle will have at least two nodes, and by our bipartite matching, the cycles are vertex-disjoint, i.e. there are no self-loops. Thus if we choose one representative node from each cycle, there can be at most half of the total number of nodes in the graph. In addition, the optimum tour traversing only these representative nodes must cost less than or equal to the original optimum because the edges all satisfy the triangle inequality. Requiring that our tour visit other nodes can only increase the cost.
- (c) If we repeat this finding and selecting representatives from minimum cycle covers, we at least halve the number of nodes in each step until we get down to one node. This clearly takes $O(\log n)$ steps. When we unravel the selections we’ve done, we essentially take the cycle represented by each represented node. This is clearly a tour of the whole graph because the cycle covers in each stage must cover every vertex and the final tour we take starts and ends at the same node. Since the cost of the minimum cycle cover of a set of nodes is a lower bound on the cost of the optimum tour over the same nodes, which can be at most OPT in any of the steps we took, we have a total cost of at most $O(\log n)OPT$.

- Problem 5.** (a) Let S be the optimum scheduling of the jobs allowing preemption, and suppose that S does not follow the greedy strategy of taking the available job j with shortest remaining processing time.

Let t be the first time at which an available job j_1 is scheduled with remaining processing time p_{j_1} in S , but there is an available job j_2 with remaining processing time p_{j_2} , such that $p_{j_2} < p_{j_1}$.

Consider the following schedule S' : we schedule j_2 at time t , and after it has completed, schedule jobs in the order according to S . We argue that the average completion time S' is less than the completion time of S . Let the completion times of jobs under S' be C'_i , and let the completion times of jobs under S be C_i . Consider the completion time C_{j_2} of job j_2 in schedule S . Let I be the set of jobs in schedule S that completed between time t and C_{j_2} (excluding job j_2). Consider the set of all jobs J . Then, we have that:

$$\forall j \in J \setminus I : C_j = C'_j.$$

That is, the only jobs whose completion times have changed are those in the set I .

The completion time of job j_2 under schedule S' is $t + p_{j_2}$. Furthermore, for every job $j \in I$, we have:

$$C'_j \leq C_j + p_j$$

where p_j is the remaining processing time for job j at time t . Furthermore, since each job in J (as well as job j_2) completes before time C_{j_2} in schedule S , we have:

$$p_{j_2} + \sum_{j \in J} p_j \leq C_{j_2} - t$$

We want to show that $\sum C_j > \sum C'_j$:

$$\begin{aligned} \sum C_j - \sum C'_j &= \left(C_{j_2} + \sum_{j \in I} C_j \right) - \left(C'_{j_2} + \sum_{j \in I} C'_j \right) \\ &\geq \left(C_{j_2} + \sum_{j \in I} C_j \right) - \left(C'_{j_2} + \sum_{j \in I} C_j + p_{j_2} \right) \\ &= C_{j_2} - C'_{j_2} - \sum_{j \in I} p_{j_2} \\ &> C_{j_2} - C'_{j_2} - \sum_{j \in I} p_j \\ &\geq t + p_{j_2} - C'_{j_2} \\ &\geq 0 \end{aligned}$$

- (b) Let S' be the nonpreemptive scheduling of the jobs according to their completion times in the optimal preemptive schedule S . Let the completion times in S' be C'_j and let the completion times in S be C_j . Note that job j and all the earlier jobs in S' must be released before time C_j to have been feasible in the preemptive schedule. In addition, the sum of those processing times is less than or equal to C_j . Job j can always be scheduled at $C_j - p_j$ at the latest because if we simply schedule all of the previous jobs in consecutive order starting at C_j (which must be feasible), job j ends at time at most $C_j + C_j = 2C_j$, and we have the desired result.
- (c) The optimal average completion time of non-preemptive scheduling is a lower bound on the average completion time of preemptive scheduling. Our algorithm is to compute the optimal pre-emptive schedule, and then use part (b). This yields a 2-approximation.