Massachusetts Institute of Technology
6.854J/18.415J: Advanced Algorithms
David Karger

Handout **??**
October 31, 2012

Problem Set 8 Solutions

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

> Arbitrarily pick a point $p \in S$, and let $C = \{p' | d(p, p') \leq d\}$. 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 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)** For any edge (u, v) in the matching found in OPT, there must be at least one edge in the greedy matching that contains u or v or otherwise the greedy matching would certainly have been able to consider (u, v) and picked it. Thus at the very least, the greedy matching includes half of the vertices as the optimal matching and thus must include at least half the number of edges as OPT. It is a proof mistake to only look at an edge in the greedy matching and claim that it corresponds to at most two edges in OPT because it does not account for the case where there are edges in OPT that do not correspond to any edges in the greedy matching (though a simple argument will avoid this problem).

This strategy is easily implemented by keeping all the edges in an unordered set data structure. In each iteration, pick an edge (u, v) from the set and remove all edges incident to u or v. Continue until the set is empty. This takes $O(m + n)$ time.

(b) Once again, consider an edge (u, v) of weight w in OPT. We argue as before that either u or v must have had an incident edge in the greedy matching. One of these incident edge(s) must have weight greater than or equal to w, or otherwise the greedy algorithm would have considered and picked (u, v) first. Call these edge(s) *heavy*. A heavy edge corresponds to at most two optimal edges of weight less than or equal to it. The total weight of the heavy edges of the greedy matching (and thus the total weight of the entire greedy matching) must therefore be at least half the weight of OPT.

To implement this algorithm, keep all the edges in a priority queue sorted by their weights. In each iteration, pick the edge (u, v) of maximum weight from the queue, and delete all edges incident to u or v. Repeat until the queue is empty. Using Fibonacci heaps, this can all be done in $O(m \log m)$ or $O(m \log n)$ time, because there are $\leq m$ entries in the queue.

(c) Nothing in the argument in part (b) relied on bipartiteness, so the greedy strategy finds a 2-approximation in the general case.

Problem 3. **(a)** Let T be the optimum Steiner tree in G, with cost OPT. Pick an arbitrary terminal node v in T, and consider the Euler tour T_E that begins at v. Clearly, the cost of this Euler tour is at most 2OPT. The Euler tour induces an order π on the terminal nodes visited.

Consider the path π through G'. This is a spanning tree T', since it visits all the nodes in G'. Furthermore, it is of cost no more than the cost of the Euler tour; to see this, consider an edge (u, v) in G'. The cost of this edge in G' is the cost of the shortest path in G. But the cost of the path from u to v in the Euler tour can be no less than this. It follows that the cost of T' is at most the cost of T_E, which is at most 2OPT. The cost of T' is an upper bound on the cost of the MST on G'.

(b) The approximation algorithm is as follows: first, compute the graph G'. Then find the MST T' on G'. For each edge $e \in T'$, let $f(e)$ be the set of edges on the shortest path in G that it represents. Consider the subgraph $T = \cup_{e \in T'} f(e)$; it spans all the terminal nodes. By part (a), the cost of T is at most 2OPT. Note that T is not necessarily a tree; however, by removing edges in cycles in T, we can obtain a Steiner tree.

Problem 4. **(a)** Choose an arbitrary vertex, and color it red. Then color all of its neighbors blue, then all of their neighbors red, and so on. Do this for each connected component of the graph. If the graph is 2-colorable, this will give a solution; every color except the first is uniquely determined by the colors chosen before it.

(b) We pick an arbitrary vertex, and color it with some color not yet chosen for any of its neighbors. This is always possible by Pigeonhole, since since there are $d+1$ colors, but only d neighbors. Continuing in this way, we can color the entire graph.

(c) We use the following algorithm:

While there exists some vertex v with degree greater than $\sqrt{n}$, we color it and its neighbors in the following way: choose an arbitrary color for v. Now, since we are given that the graph is 3-colorable, all of the subgraph consisting of the neighbors of v must be 2-colorable. Thus we can 2-color them in polynomial time using two other arbitrary colors. Now, we have colered at least $\sqrt{n}$ vertices. We remove them from the graph. The three colors that we used will not be used again.

Once this process completes, we will have used at most $3(n/\sqrt{n}) = O(\sqrt{n})$ colors. If any vertices are left, then each vertex has degree at most $\sqrt{n}$ (otherwise, the above procedure would keep running). Thus by (b), we can color them using at most $\sqrt{n}+1$ additional colors. Therefore, in total, this method uses at most $O(\sqrt{n})$ colors.

Problem 5. **(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

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

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:

$$\begin{array}{rcl} W - (1-\delta)W_A & \leq & (1+\epsilon)W_B \\ & \Longleftrightarrow & \\ \delta & \leq & \dfrac{(1+\epsilon)W_B - W + W_A}{W_A} \\ & \Longleftrightarrow & \\ \delta & \leq & \dfrac{\epsilon W_B}{W_A} \end{array}$$

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 $\frac{n}{\epsilon OPT}$ 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 6. **(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.