P1:10
P2:10
P3:10
P4:10
P5:10

6.854
Advanced Algorithms
2004/10/26

Dan Ports
drkp@mit.edu
Collaborators: {sarahl}

# Problem Set 7

**Problem 1:**

**a)** Suppose $G$ has an independent set of size $k$. We show that the product graph (call it $G^*$) has an independent set of size $k^2$. Note that for each vertex $u$ and $v$ in the independent set, there is no edge $(u, v)$ in the graph, so the vertices of the subgraphs $G_u$ and $G_v$ have no edges between them. If we take the $k$ vertices of the original independent set, we have $k$ independent subgraphs. Each subgraph contains an independent set of size $k$: it is the same independent set as the original graph, since these are copies of that graph. These $k$ independent sets of size $k$ form an independent set of size $k^2$, since there are no edges between them.

**b)** Suppose $G^*$ has an independent set of size $s$. We show that $G$ must have an independent set of size $\sqrt{s}$. Consider first the case in which the independent set in $G^*$ contains $\sqrt{s}$ vertices in the same subgraph. These vertices form an independent set of size $\sqrt{s}$, so $G$ contains the same independent set since each subgraph is a copy of it.

Now suppose $G^*$'s independent set does not contain $\sqrt{s}$ vertices in the same subgraph. Since the independent set has size $s$, and each subgraph contains less than $\sqrt{s}$ vertices, there must be more than $\sqrt{s}$ subgraphs that contain vertices in the independent set. This means that, for each such subgraph $G_v$, an independent set can be formed in $G$ by taking the vertices $v$, by definition of the product graph. This independent set has size at least $\sqrt{s}$. This proves the claim.

**c)** Suppose that we have a $\alpha$-approximation for the independent set problem. We show that for any $0 < \epsilon < 1$, we can find an $\epsilon$-approximation in polynomial time. We choose $n$ large enough that

$$\alpha^{(1/2)^n} > \epsilon$$

then compute a graph $G'$ by taking the graph product of $G$ $n$ times. We then use the independent set approximation algorithm on $G'$. This graph has an independent set of size $OPT^{2^n}$, so the algorithm returns a set of size $\alpha OPT^{2^n}$. By repeatedly applying part $b$, we find an independent set in $G$ that has size

$$\alpha^{2^n} (OPT)^{(1/2)^n (2)^n} > \epsilon OPT$$

which is the desired result.

The algorithm's running time is polynomial in the size of $G$ since it simply requires computing the product graph $n$ times, and performing the $\alpha$-approximation, and $n$ is a function only of $\epsilon$, not $|G|$.

**Problem 2:**

**a)** Our algorithm is as follows: as long as there are points that have not been assigned to a cluster, we select any point, and make it a cluster center, and assign all the points within distance $d$ from it to the cluster.

The clusterings produced by this algorithm all have diameter less than $2d$: the distance from any point to the center is less than $d$, so by the triangle inequality the distance from any point to any other point must be less than $2d$.

To show that this gives a valid 2-approximation for $k$-clustering, we must also show that it produces at most $k$ clusters. Consider any cluster that is added by the greedy algorithm. The center point $p$ of this cluster is contained within some cluster in the optimal solution. The distance from $p$ to every other point in the optimal cluster is less than the diameter of the cluster, which is $d$. But every point within distance $d$ of $p$ is in the greedy cluster, so the optimal cluster is contained within the greedy cluster. There are at most $k$ optimal clusters, and each greedy cluster contains at least one optimal cluster, so there are at most $k$ clusters in the greedy solution.

**b)** Suppose we choose our $k$ centers as the points at maximum distance from all previously chosen centers, and assign all points to the nearest center. We show that this gives a 2-approximation. All points are assigned to the nearest center, which is no further than distance $d$. For purposes of contradiction, suppose there is some point $p$ that is greater than distance $d$ from the nearest center. Note that in this case, all centers will also be at least distance $d$ away from each other. But there are at most $k$ clusters, so either 2 centers or one center and $p$ will be in the same cluster in the optimal solution. But these two points have distance at least $d$, which contradicts the definition of $d$ as the optimal cluster distance. So no such point $p$ can exist. Thus, since all points are within $d$ of the nearest center, the previous result shows that the clusters defined by the centers have diameters at most twice as large in diameter as optimal. This proves the result.

**Problem 3:**

**a)** Suppose $G'$ is the graph of all terminals in $G$, where edge weights are given by the shortest path distance between the endpoints in $G$. We show that the cost of the minimum spanning tree in $G'$ is less than twice cost of the optimal Steiner tree in $G$.

Let $T$ be a Steiner tree in $G$. Define $T'$ to be the same as $T$, but with two copies of each edge. Then consider an Euler tour on $T'$. This is a tour that passes through each terminal, since each terminal is on the Steiner tree. So we can translate this into an equivalent tour on $G'$ that travels from one terminal to the next in the same order as the tour on the Steiner tree. The tour on $G$ has cost precisely twice that of the Steiner tree, and the tour on $G'$ has at most the same cost, since the tour is a sequence of paths between each terminal in $G$, and the tour in $G'$ is a sequence of *shortest* paths between each terminal. Note that if we remove some edge from the tour in $G'$ (which only decreases the cost), we convert the tour into a spanning tree on $G'$. The cost of this tree must be at least that of the minimum spanning tree, and it is also less than twice the cost of the Steiner tree. So the cost of a minimum spanning tree in $G'$ is less than twice the cost of the optimum Steiner tree.

**b)** This gives a 2-approximation algorithm for the Steiner tree problem. Given any graph $G$, we compute the all-pairs shortest path distances using the Floyd-Warshall algorithm. We use these distances to construct the shortest-path graph $G'$. We then find a minimum spanning tree of $G'$ using Prim's algorithm (perhaps using Fibonacci heaps, because everyone likes Fibonacci heaps). We then convert the MST on $G'$ back into a Steiner tree on $G$. To do so, we replace every edge in the MST with the shortest path between the endpoints in $G$ (using each edge in $G$) at most once. This gives a Steiner tree for $G$, and from above it is a 2-approximation of the optimal Steiner tree.

**Problem 4:**

**a)** Suppose preemption of jobs is allowed. We show that the greedy algorithm of working on the released job with the shortest remaining time is optimal. Suppose $O$ is an optimal solution. We show it must also be a greedy solution. First, if any job is scheduled over an interval which contains

the release time of another job, for simplicity of analysis, we divide it into two pieces of the same job delimited by the release time. Next, suppose $O$ is not a solution produced by the greedy algorithm. Then there must be some (piece of a) job $j_1$ that is scheduled immediately before another job $j_2$, but the remaining completion time of job $j_2$ is shorter than that of $j_1$ at the time that work began on $j_1$. Let $r_1$ be the remaining time for job $j_1$, and analogously $r_2$ for $j_2$. So from that point, it takes $r_1$ time to complete $j_1$ and $r_1 + r_2$ time to complete $j_2$, i.e. $2r_1 + r_2$ total for the two. If we swap them, it will take $r_2$ time to complete $j_2$ and $r_1 + r_2$ time to complete $j_2$, i.e. $r_1 + 2r_2$ time total for the two. Since the remaining completion time of job $j_2$ is shorter than that of $j_1$, $r_2 < r_1$, and so the total completion time decreases after we make the swap. Thus, the average completion time decreases. This contradicts the optimality of $O$. So the greedy algorithm produces the optimal result.

**b)** Suppose we generate a non-preemptive schedule by scheduling all jobs in the order they are completed in the optimal preemptive schedule. This leads to idle time in the schedule when the next job to be scheduled has not been released yet. We show that introducing this idle time only increases the completion length of each job by a factor of two.

Consider a job $x$. $x$'s completion time $c'_x$ is greater than its completion time $c_x$ in the preemptive schedule only by the amount of added idle time that takes place before $x$ is processed, since the jobs are still completed in the same order. Recall that idle time is added only if a job is preempted, and the amount of idle time added for a job $y$ cannot be more than the processing time $p_y$, since otherwise $y$ could be completed earlier. So the completion time of job $x$ increases by the total amount of idle time added for all jobs before $x$. But these jobs complete before $x$, so their total processing time is at most the completion time $c_x$ of $x$. So $c'_x \leq 2c_x$. This proves the claim.

**c)** Since the completion time for each job at most doubles, the average completion time over all jobs at most doubles.

We use this to give a 2-approximation for this scheduling problem. We first use the greedy algorithm to generate a preemptive schedule (in polynomial time), then generate a non-preemptive schedule by scheduling the jobs in order of their completion time in the preemptive schedule. This requires linear time in the length of the schedule, so it is also polynomial. The average completion time is increased by at most a factor of 2, so this is a 2-approximation.

**Problem 5:**

**a)** Suppose there are at most $k$ distinct item sizes. We can find the optimal bin-packing in polynomial time by enumeration. We use the same argument as for the $P||C_{max}$ approximation: note that there are a maximum of $n^k$ possible profiles of the input. We enumerate the set $Q$ of profiles $\langle x_1, x_2, \ldots, x_k \rangle$ that will fit in a single bin. We let $M(\langle x_1, x_2, \ldots, x_k \rangle)$ be the minimum number of bins required to bin-pack a profile, and note that $M(q) = 1$ for all $q \in Q$. Then we use the dynamic program recurrence

$$M(\langle a_1, a_2, \ldots, a_k \rangle) = 1 + \min_{\langle x_1, x_2, \ldots, x_k \rangle \in Q} M(\langle a_1 - x_1, a_2 - x_2, \ldots, a_k - x_k \rangle)$$

The size of the dynamic program (as it is in $P||C_{max}$) is $O\left(n^{2k}\right)$ which is polynomial in $n$.

By computing the dynamic program and maintaining backpointers as we use the recurrence, we can find the minimum number of bins $M(\langle x_1, x_2, \ldots, x_k \rangle)$ required for an input with profile $\langle x_1, x_2, \ldots, x_k \rangle$, and obtain the optimum bin-packing.

**b)** Suppose all items of size greater than $\epsilon$ have been packed into $B$ bins. We can pack the remaining items in linear time by placing each item into the most empty bin, or adding a new bin if it will not fit into any existing bin.

This requires $\max\{B, 1 + (1 + O(\epsilon))B^*\}$ bins. If there are no items of size less than $\epsilon$, or all items of size $\epsilon$ can be packed into the empty space in the $B$ bins, then $B$ bins are required. If new bins are needed, then call the number of bins required $B'$. All except the last bin are full enough that no other items could be added, i.e. they contain items totaling $1 - \epsilon$ in size. So the total size of all items in these bins must be more than $(B' - 1)(1 - \epsilon)$, and so $B^* > B'(1 - \epsilon) - 1$, or

$$B' \leq \frac{B^*}{1 - \epsilon} \leq B^*(1 + 2\epsilon) + 1$$

since for sufficiently small $\epsilon$, $1 + 2\epsilon \leq \frac{1}{1-\epsilon}$ $(0 < \epsilon < 1/2)$. This proves the bound.

**c)** We cannot round item sizes up to the nearest power of $(1 + \epsilon)$, since $(1 + \epsilon)$ is greater than 1, and items of size greater than 1 cannot be packed into unit-size bins.

**d)** For fixed $k$, order the items by size, with $S_1$ being the largest $n/k$, $S_2$ being the next largest $n/k$, etc. We increase the size of the items to the largest size in its set. Remove $S_1$ from the bins. We claim that the remaining items will still fit in the same bins. Note that the largest item in $S_2$ was smaller than every item in $S_1$, so even after rounding every item in $S_2$ to the largest size, we can still fit it in the spaces that were used for the $S_1$ items in the original packing. The same is true for moving $S_3$ into the space previously used by $S_2$, and so on. Thus, all of the items except those in $S_1$ can be packed into the original bins. Adding back the $S_1$ requires at most $n/k$ more bins, since in the worst case each item in $S_1$ now requires its own bin.

**e)** Suppose that we have a set of items of any size. We first consider all items of size greater than $\epsilon/2$. We group these items into $\frac{2}{\epsilon^2}$ groups, and use the rounding procedure defined above to limit the number of distinct sizes to one per group, then use the dynamic programming approach from part $a$ to find the optimal bin-packing on the rounded set. The result is a bin-packing that exceeds the optimum by at most $\frac{n\epsilon^2}{2}$. Since the optimum is at most $n\frac{\epsilon}{2}$, our packing is a $(1 + \epsilon)$ approximation of the optimum.

We then pack the remaining items (of size less than $\epsilon/2$) using the greedy linear-time packing algorithm. Here, $B = (1 + \epsilon)B^*$, so this achieves a packing that requires at most $1 + (1 + \epsilon)B^*$ bins, which is the desired bound. The algorithm requires polynomial time, so it is an asymptotic polynomial time approximation scheme.

**Problem 6:**

About 12 hours.
This is polynomial in the problem length, but only because I used an approximation algorithm.