Introduction to Algorithms	Problem Set 10
CS 4820, Spring 2014	Due 11:59pm Monday, May 5

Hand in your solutions electronically using CMS. Each solution should be submitted as a separate file. For multi-part problems, all parts of the solution to that problem should be included in a single file.

Remember that when a problem asks you to design an algorithm, you must also prove the algorithm's correctness and analyze its running time. The running time must be bounded by a polynomial function of the input size. The same guideline applies to reductions.

1. *(15 points in total)* Recall that in the Knapsack Problem, one is given a set of items numbered $1, 2, \ldots, n$, such that the i-th item has value $v_i \geq 0$ and size $s_i \geq 0$. Given a total size budget B, the problem is to choose a subset $S \subseteq \{1, 2, \ldots, n\}$ so as to maximize the combined value, $\sum_{i \in S} v_i$, subject to the size constraint $\sum_{i \in S} s_i \leq B$.

1.a. *(5 points)* Consider the following *greedy algorithm*, GA.

1. Eliminate items whose size s_i is greater than B.
2. For each remaining i, compute the *value density* $\rho_i = v_i/s_i$.
3. Sort the remaining items in order of decreasing ρ_i.
4. Choose the longest initial segment of this sorted list that does not violate the size constraint.

Also consider the following *even more greedy algorithm*, EMGA.

1. Eliminate items whose size s_i is greater than B.
2. Sort the remaining items in order of decreasing v_i.
3. Choose the longest initial segment of this sorted list that does not violate the size constraint.

For each of these two algorithms, give a counterexample to demonstrate that its approximation ratio is not bounded above by any constant C. (Use different counterexamples for the two algorithms.)

1.b. *(5 points)* Now consider the following algorithm: run GA and EMGA, look at the two solutions they produce, and pick the one with higher total value. Prove that this is a 2-approximation algorithm for the Knapsack Problem, i.e. it selects a set whose value is at least half of the value of the optimal set.

1.c. *(5 points)* Use part (b) to show that the the dynamic-programming based approximation algorithm for Knapsack presented in class can be modified so that it achieves an approximation ratio of at most $1 + \varepsilon$ with running time $O(n^2/\varepsilon)$. In your solution, it is not necessary to repeat the proof of correctness of the algorithm presented in class.

Solution

1.a. A counterexample for GA is given by an instance with two elements: $(v_1 = 2, s_1 = 1)$ and $(v_2 = 2C + 1, s_2 = 2C + 1)$. If the knapsack capacity is $2C + 1$, then GA will pick the first element and then the second one won't fit. The value obtained will be only 2, whereas value $2C + 1$ would be obtained by taking the second element instead.

A counterexample for EMGA is given by an instance with $2C + 2$ elements: one element with $(v_1 = 2, s_1 = 2C + 1)$ and $2C + 1$ elements each with $(v_i = 1, s_i = 1)$. If the knapsack capacity is $2C + 1$, then EMGA will pick the first element and then all the remaining ones won't fit. The value obtained will be only 2, whereas value $2C + 1$ would be obtained by taking all elements except the first one.

1.b. First of all, let's assume that there are no items whose size is greater than B. Such items will never appear in any solution, and can safely be eliminated without changing the value of the optimum nor the behavior of either of the greedy algorithms under consideration.

Under this assumption, we'll prove that

$$\text{GA} + \text{EMGA} \geq \text{OPT} \tag{1}$$

from which is follows that $\max\{\text{GA}, \text{EMGA}\} \geq \frac{1}{2}\text{OPT}$. To prove (1), let $i_1, \ldots, i_k$ denote the sequence of items selected by GA (in the order selected) and let i_{k+1} be the next element that GA would have selected, if it hadn't run out of space. The EMGA algorithm gets a value at least as high as the value of item i_{k+1}, and the value of items $i_1, \ldots, i_k$ is, of course, exactly the value achieved by the GA algorithm. On the other hand, the combined value of items $i_1, \ldots, i_k, i_{k+1}$ is greater than that of the optimal knapsack solution S_{OPT}, since S_{OPT} has a strictly smaller combined size, and any elements of S_{OPT} that are not among $\{i_1, \ldots, i_{k+1}\}$ have a value density that is strictly smaller than the least dense element of that set. In more detail, let

$$\begin{aligned} S_0 &= S_{OPT} \cap \{i_1, \ldots, i_{k+1}\} \\ S_1 &= \{i_1, \ldots, i_{k+1}\} \setminus S_{OPT} \\ S_2 &= S_{OPT} \setminus \{i_1, \ldots, i_{k+1}\} \end{aligned}$$

If we denote the value density of each item i by $\rho_i = v_i/s_i$, and we let $\rho^* = \rho_{i_{k+1}}$, then by the definition of the greedy algorithm we know that $\rho_{i_j} \geq \rho^*$ for $j = 1, \ldots, k+1$ whereas $\rho_i \leq \rho^*$ for every $i \notin \{i_1, \ldots, i_{k+1}\}$. Consequently $v_i \geq \rho^* s_i$ for every $i \in S_1$, while $v_i \leq \rho^* s_i$ for every $i \in S_2$. This implies that

$$\begin{aligned} \left(\sum_{j=1}^{k+1} v_{i_j}\right) - \left(\sum_{i \in S_{OPT}} v_i\right) &= \left(\sum_{i \in S_1} v_i\right) - \left(\sum_{i \in S_2} v_i\right) \\ &\geq \left(\sum_{i \in S_1} \rho^* s_i\right) - \left(\sum_{i \in S_2} \rho^* s_i\right) \\ &= \rho^* \left(\sum_{i \in S_1} s_i - \sum_{i \in S_2} s_i\right). \end{aligned} \tag{2}$$

The right side of (2) is positive, since

$$\sum_{i \in S_1} s_i > B - \sum_{i \in S_0} s_i \geq \sum_{i \in S_2} s_i.$$

Hence, the left side of (2) is also positive, i.e.

$$v_{i_1} + \cdots + v_{i_{k+1}} > \text{OPT}, \tag{3}$$

which establishes (1).

1.c. In class, Professor Kleinberg presented a dynamic programming algorithm for the knapsack problem that runs in time $O(nv^*)$, where v^* denotes an upper bound on the value of the optimum solution. (The algorithm is also presented in Chapter 11.8 of the book.) The reason this isn't a polynomial-time algorithm is that the running time is proportional to v^*, which may be exponential in the input size since values in the input are specified as binary integers.

To obtain an approximation algorithm whose running time doesn't depend linearly on v^*, let us precompute a set S_* obtained by running GA and EMGA to obtain two sets, and taking the one with the higher total value. Let $v_* = \sum_{i \in S_*} v_i$ and let

$$k = \frac{\varepsilon v_*}{(1+\epsilon)n}.$$

Furthermore, for every item i let

$$\hat{v}_i = \left\lfloor \frac{v_i}{k} \right\rfloor.$$

Let S denote the output of the dynamic program on the instance defined by the value-size pairs $\{(\hat{v}_i, s_i) \mid i = 1, \ldots, n\}$, and let S_{OPT} be the optimal set for the actual value-size pairs $\{(v_i, s_i) \mid i = 1, \ldots, n\}$. Since the dynamic program outputs an optimal solution for the knapsack instance defined by $\{(\hat{v}_i, s_i)\}$, we know that

$$\begin{aligned}
\sum_{i \in S} \hat{v}_i &\geq \sum_{i \in S_{OPT}} \hat{v}_i \\
\sum_{i \in S} v_i &\geq \sum_{i \in S} k\hat{v}_i \geq \sum_{i \in S_{OPT}} k\hat{v}_i \geq \sum_{i \in S_{OPT}} (v_i - k) \\
\sum_{i \in S} v_i &\geq \left(\sum_{i \in S_{OPT}} v_i \right) - nk = \left(\sum_{i \in S_{OPT}} v_i \right) - \frac{\varepsilon v_*}{1+\varepsilon} \\
&\geq \left(1 - \frac{\varepsilon}{1+\varepsilon}\right) \left(\sum_{i \in S_{OPT}} v_i \right) \qquad \text{(since } v_* \leq \textstyle\sum_{i \in S_{OPT}} v_i\text{)} \\
&= \frac{1}{1+\varepsilon} \sum_{i \in S_{OPT}} v_i
\end{aligned}$$

hence the value of S is a $(1+\varepsilon)$-approximation to the value of S_{OPT}.

Substituting the specified value of k into the running time of the dynamic program, we get $O(n^2(v^*/v_*)/\varepsilon)$. Recalling the definition of v_*, we see that part (b) of this exercise already established that $v^*/v_* \leq 2$, so the overall running time of the dynamic program is $O(n^2/\varepsilon)$. The preprocessing step of computing the value density of every element and running GA and EMGA to determine the set S_* takes only $O(n \log n)$ time, so the overall running time of the algorithm is $O(n^2/\varepsilon)$.

2. *(10 points in total)*

2.a. *(5 points)* Consider the special case of unweighted set cover in which every element belongs to at most k sets. Design a polynomial-time $O(k)$-approximation algorithm for this problem.

2.b. *(5 points)* Consider the same specialization of weighted set cover and design a polynomial-time $O(k)$-approximation algorithm for it. (If you are confident in your solution of this part, you can simply write, "Special case of 2b" as your solution to 2a. In that case we'll simply double your score for this part of the problem so that it counts for 10 points instead of 5.)

Solution

The case $k = 2$ is essentially equivalent to the Vertex Cover problem. Interpret elements as edges of a graph and sets as vertices. A set contains an element if that vertex is an endpoint of that edge. The fact that every element belongs to at most two sets corresponds to the fact that an edge of a graph has

at most two endpoints, and the only reason the correspondence between the two problems isn't precise is that an edge of a graph always has *exactly* two endpoints, whereas an element in the $k = 2$ case of our problem can either belong to one or two sets. Despite this slight discrepancy between Vertex Cover the $k = 2$ case of this problem, we will base our solutions to both 2.a and 2.b on the corresponding approximation algorithms for unweighted and weighted vertex cover that were taught in class.

2.a. The algorithm is very simple.

```
Mark every element as uncovered.
while there exists an uncovered element x do
    Select every set containing x.
    Mark every element in the union of those sets as covered.
end while
```

Assume that the input is represented by an array of n elements, where each entry in the array is a list of the names of at most k sets containing that element. (If the input is represented in some other format, it only takes polynomial time to construct the desired input format.) A single iteration of the **while** loop requires at most $O(n)$ time to check for uncovered elements, $O(k)$ time to find all of the sets containing this element, and $O(kn)$ time to run through the lists of sets containing every other element and mark those that belong to one of the newly selected sets. Thus, one while-loop iteration takes $O(kn)$ time. As there are at most n iterations, the running time of the algorithm is $O(kn^2)$.

Let t be the number of iterations of the **while** loop, and let $i_1, \ldots, i_t$ be the elements chosen in those loop iterations. Note that no set contains more than one of the elements $i_1, \ldots, i_t$; indeed if any set contained i_p and i_q for some $p < q$, then that set would have been selected in the p^{th} iteration of the **while** loop and all of its elements, including i_q would have been marked as covered at that time, contradicting the fact that i_q was uncovered in a later iteration. Thus, no set covers more than one of $i_1, \ldots, i_t$, which implies that an optimal solution must use at least t sets. In comparison, our algorithm chooses at most k sets in each of its t **while** loop iterations, for a total of at most kt sets. This completes the proof that the number of sets chosen by the algorithm is at most k times the optimum, i.e. that the algorithm is a k-approximation algorithm.

2.b. The algorithm is based on the "pricing method", also known as the primal-dual method. As stated earlier in this solution, it is adapted from the corresponding approximation algorithm for Weighted Vertex Cover. We will assume that the elements are numbered $1, 2, \ldots, n$ and that the sets are numbered $1, 2, \ldots, m$. The list of all sets containing element i will be denoted by L_i.

```
Initialize S = {j | w_j = 0}, y_i = 0 ∀i ∈ [n], s_j = 0 ∀j ∈ [m].
for all i ∈ [n] do
    δ = min{w_j − s_j | j ∈ L_i}.
    y_i = y_i + δ
    for all j ∈ L_i do
        s_j = s_j + δ
    end for
    S = S ∪ {j ∈ L_i | s_j = w_j}
end for
return S
```

The running time is $O(kn)$: the outer loop iterates n times, the inner loop iterates at most k times within each outer loop iteration, and only $O(1)$ work is done within an inner loop iteration. (Additionally, at most $O(k)$ extra work is done within an outer loop iteration, when computing the value of δ.)

To prove the approximation ratio, we state the following invariants which hold at the end of each iteration of the outer loop. We omit the easy inductive proofs that these invariants continue to hold at the end of each outer loop iteration.

1. For each $j \in [m]$, $s_j = \sum_{i \in T_j} y_i$, where T_j denotes the set of all $i \in [n]$ such that $j \in L_i$.
2. For each $j \in [m]$, $s_j \leq w_j$.
3. $S = \{j \mid s_j = w_j\}$.

We now compare the weights of our set S and of the optimum set cover S^* with the quantity $\sum_{i=1}^{n} y_i$. First, observe that S^* is a set cover, so for all $i \in [n]$ we have $|S^* \cap L_i| \geq 1$. Hence,

$$\sum_{j \in S^*} w_j \geq \sum_{j \in S^*} s_j = \sum_{j \in S^*} \sum_{i \in T_j} y_i = \sum_{i=1}^{n} \sum_{j \in S^* \cap L_i} y_i = \sum_{i=1}^{n} y_i \cdot |S^* \cap L_i| \geq \sum_{i=1}^{n} y_i.$$

On the other hand,

$$\sum_{j \in S} w_j = \sum_{j \in S} s_j \leq \sum_{j=1}^{m} s_j = \sum_{j=1}^{m} \sum_{i \in T_j} y_i = \sum_{i=1}^{n} \sum_{j \in L_i} y_i = \sum_{i=1}^{n} y_i \cdot |L_i| \leq k \sum_{i=1}^{n} y_i.$$

Combining these two inequalities, we find that

$$\sum_{j \in S} w_j \leq k \sum_{i=1}^{n} y_i \leq k \sum_{j \in S^*} w_j,$$

which completes the proof that our algorithm is a k-approximation algorithm.

3. *(10 points in total)* You are asked to create a schedule for the upcoming season of a sports league. Based on records from the previous year, it was decided which pairs of teams should play against each other in the upcoming season. Games are played on Sundays and each team can play at most one game on the same Sunday. The goal is to schedule all games in such a way that the number of weeks is minimized.

This scheduling problem is equivalent to the following problem on graphs (called EDGE COLORING): You are given a graph G with vertices $t_1, \ldots, t_n$ corresponding to the teams. The edges of the graph correspond to the set of games that are to be scheduled in the upcoming season. An *edge coloring* of G assigns to each edge (t_i, t_j) in the graph a "color" w_k such that no vertex is incident to more than one edge of the same color. (This constraint ensured that all games (t_i, t_j) with the same color w_k can be held in the same week.) Your goal is to find an edge coloring of G that uses as few colors as possible.

Give an efficient algorithm to compute an edge coloring of G that uses at most $2d - 1$ different colors, where d is the maximum degree in the graph. Show that the approximation ratio of this algorithm is at most 2.

Solution

We assign colors in a greedy way. We start with all edges uncolored. As long as there exists an uncolored edge e, we assign a color to e that is not present at any of the two endpoints of e. The number of edges that touch one of the two endpoints of e is at most $2d-2$ if we exclude e. Hence, if we have $2d-1$ colors available, there exists at least $(2d-1) - (2d-2) = 1$ color that we can assign to e without creating a conflict. The running time is $O(md)$, since for each edge we need $O(d)$ steps to enumerate all of the colors that are already used on adjacent edges and to find an available color for e.

The approximation ratio of this algorithm is at most 2. If v is a vertex with degree d, then every valid edge coloring assigns pairwise distinct colors to the edges incident to v. Hence, the coloring uses at least d different colors. Thus, the number of colors used by the greedy algorithm is within a factor of 2 of the minimum number of colors in an edge coloring.