15-451/651 Algorithms, Spring 2019

Homework #6 **Due: April 9, 2019**

(25 pts.) 1. **Sudoku is so Old-School.** In *Generalized Sudoku* we are given n items, a positive integer k, and m subsets $S_1, S_2, \ldots, S_m$ of these items such that $|S_i| = k$ for all i. The goal is to give each item a label from $\{1, \ldots, k\}$ so that every subset S_i contains exactly one item of each label. (So, in standard Sudoku, the items are arranged in a 9×9 grid, the sets S_i are the rows, columns and 3×3 mini-grids, and $k = 9$.) Specifically, let's define the *decision version* of Generalized Sudoku to be: given the sets $S_1, S_2, \ldots, S_m$ and the integer k, does a solution of the desired form exist? The *search version* of the problem is to actually find a solution of the desired form, if one exists.

(a) Prove that the decision version of Generalized Sudoku is in NP.

(b) Prove that the decision version of Generalized Sudoku is NP-hard for every $k \geq 3$. (You may not use that $n \times n \times n$-Sudoku is NP-complete, as claimed in class. Please reduce from some other problem mentioned in lecture or in recitation.)

Solution: (a) First, we reduce the 3-coloring problem to Sudoku with $k = 3$. Specificaly, given a graph $G = (V, E)$, let the items be $V \cup E$, and for each $e = (u, v) \in E$ let $S_e = \{u, v, e\}$ (i.e. for each edge, create a set containing the edge and its two endpoints). If G has a 3-coloring, say using colors $\{1, 2, 3\}$, then by giving each edge the label that is not used by its endpoints we have a legal solution to the Sudoku problem. In the other direction, given a solution to the Sudoku problem, the labels used on the vertices must be a legal 3-coloring of G since no edge will have both its endpoints the same color. Since we know that 3-coloring is NP-complete, this implies that Generalized Sudoku with $k = 3$ is NP-hard.

Now, we can inductively show Generalized Sudoku is NP-hard for any $k \geq 3$. Suppose we know Generalized Sodoku is NP-complete for $k-1$. Then we can reduce this to Generalized Sodoku for k by adding an extra element, k, to every set. Clearly, we can label the new sets with k labels iff the old sets could be labeled with $k - 1$.

(25 pts.) 2. **We All Have our Problems.** The instructors in your course have changed the rules for how they award credit for the course. As before, you have a set of m problems you can solve, and n chapters you can read. Problem p_j can be solved only if you have read all the chapters in set $P_j \subseteq \{1, 2, \ldots, n\}$. The cost of chapter i is an integer $c_i > 0$. So far the same. But you're also given an integer $L > 0$, and the new rule says: you get a good grade in the course if you solve at least L problems out of n. As an enterprising student, you'd like to minimize the cost of reading chapters necessary to get a good grade.

The decision problem now is: given n chapters with integer costs, m problems with associated sets of chapters, and parameters $L \in \{1, 2, \ldots, m\}$ and $C \in \{1, 2, \ldots, \sum_i c_i\}$, does there exist a subset of chapters of total cost at most C, so that you can solve at least L problems having read these chapters.

Prove that this problem is NP-complete.

Solution: First, to show this problem is in NP. Note that if I gave you a set S of chapters, you could verify that it had cost at most C (by summing up the costs $\sum_{i \in S} c_i$ and checking if this sum was at least C), and that it could be used to solve at least L problems (just ensure there are at least L problems which depend only on these chapters in S).

Now to show NP-hardness. Let us reduce from the Clique problem. Take any instance of the clique problem, which consists of an undirected graph G, and a number k, and asks if there exists a clique in G of at least k vertices. From this we want to get an instance of our problem, we will call this instance $f(G)$. We want that G is a "Yes" instance of Clique if and only if $f(G)$ is a "Yes" instance of our problem.

Good. What is this instance $f(G)$ of our problem? Make one chapter for every node in G, each with cost $c_i = 1$. Make one problem for every edge $(u, v) \in G$. The problem (u, v) depends on the two chapters u and v. Set the target L to be $\binom{k}{2}$. And the threshold C to be k. So we are asking: is it possible to read at most k chapters so that we can solve at least $\binom{k}{2}$ problems. Note that the maximum number of problems we can solve (by the way we constructed our instance) by reading k chapters is $\binom{k}{2}$, and this is possible exactly when G has a clique on k vertices. In other words, G has a clique on k vertices if and only if our instance has a "Yes" answer. This completes the NP-hardness proof.

We've shown our problem is in NP, and it is NP-hard. Hence it is NP-complete.

(25 pts) 3. **(It's in the Bag.)** Consider the knapsack problem we solved using dynamic programming in time $O(nS)$. (See Lecture #8 for definitions; assume each item has size at most S.) This is not good if S is very large.

(a) Give a dynamic-programming algorithm with running time $O(nV^*)$, where V^* is the value of the *optimal solution*. So, this would be a better algorithm if the sizes are much larger than the values.

Note: your algorithm should work even if V^* is not known in advance. You may want to first assume you are given V^* up front and then afterwards figure out how to remove that requirement.

(b) Now given an instance I of knapsack and some real $k \geq 1$, define new values $v_i' := k \cdot \lfloor \frac{v_i}{k} \rfloor$. This gives a new instance I'. Since item sizes and S remain the same, clearly the feasible solutions to I and I' are the same.

For any feasible solution, let its value in I be V, and its value in I' be V'. Show that $V \geq V' \geq V - nk$.

(c) Use part (a) to show that I' can be solved in at most $O(\frac{n^2 v_{\max}}{k})$ time.

(d) Given any knapsack instance I and a value $\epsilon \in (0, 1)$, show that setting $k := \frac{\varepsilon\, v_{\max}}{n}$ gives an algorithm that returns a feasible solution to I, has value least $(1 - \varepsilon)$ times the optimal value of I, and runs in time $O(\frac{n^3}{\varepsilon})$.

In other words, if you wanted to find a solution whose value is within 99% of the optimum value, use this algorithm with $\varepsilon = 0.01$.

You may assume that all values and sizes are integers. And that each item can fit by itself into the knapsack, i.e., $s_i \le S$ for all i. Also, k in parts (b) and (c) is allowed to be a real number, not just an integer. Finally, your algorithm should output *the set* of items to put into the knapsack to get a value at least $(1-\varepsilon)OPT(I)$.

Solution:

(a) We will create an array s where $s[m, v]$ is the minimum knapsack size necessary such that it is possible to place items of total value *at least* v into the knapsack using a subset of the first m items. (We will set $s[m, v] = \infty$ if the total value of the first m items is less than v.) The optimal value V can then be determined by $\max\{v | s[n, v] \le S\}$. We can calculate $s[m, v]$ by running a nested loop:
for $v = 1, 2, \dots$ do:
  for $m = 0, 1, \dots, n$ do:
    if $(m = 0)$ then $s[m, v] = \infty$,
    else if $(v_m \ge v)$ then $s[m, v] = \min\{s_m, s[m-1, v]\}$,
    else $s[m, v] = \min\{s_m + s[m-1, v-v_m], s[m-1, v]\}$,
breaking out of the loop once we find $s[n, v] > S$. The optimal value V is then $v-1$. Calculation of $s(m, v)$ takes a constant time for each m and v, and we calculate $(n+1)(V^*+1)$ of them, which gives total running time of $O(nV^*)$.

The only issue not yet addressed is the time to *allocate space for* the array s, but we can handle that using the doubling-trick that we used when implementing a stack as an array, and at worst we allocate a constant-factor more space than necessary.

(b) First note that any valid solution to I is a valid solution to I' and vice versa, because the item sizes and S stay the same (if they fit in one instance, they fit in the other). For any valid solution to I, the cost of including the same items in I' is equal to $\sum_i k\lfloor \frac{v_i}{k} \rfloor$. However $\sum_i k \cdot (\frac{v_i}{k} - 1) \le \sum_i k\lfloor \frac{v_i}{k} \rfloor \le \sum_i k\frac{v_i}{k}$.
It must be the case that $V' \le V$, because otherwise the items in V' would be a valid solution to I with higher cost than V. Similarly, $V' \ge V - nk$, because the optimal solution to I has a value of at least $V - nk$ in I'.

(c) We first note that $V' \le nv_{max}$, the size of our array in part a is bounded by $n^2 v_{max}$ in size. However, in I', all values are multiples of k, so we can go by multiples of k instead of checking every v. The only change we need is to the last 2 lines:
    else if $(v_m \ge vk)$ then $s[m, v] = \min\{s_m, s[m-1, v]\}$,
    else $s[m, v] = \min\{s_m + s[m-1, (vk - v_m)/k], s[m-1, v]\}$,

(d) As stated before, any valid solution to I' is also a valid solution to I.
Plugging $k = \frac{\varepsilon v_max}{n}$ into $\frac{n^2 v_{max}}{k}$, we get that the algorithm runs in time

$$O(\frac{n^2 v_{max}}{\varepsilon v_{max}/n}) = O(\frac{n^3}{\varepsilon})$$

Finally, by part b, the optimal solution to I' has value at least $V - nk = V - \varepsilon v_{max} \ge V(1-\varepsilon)$, because every item fits into the knapsack so we can always get at least v_{max}.