15-451/651 Algorithms, Spring 2018

Homework #5 **Due: April 3, 2018**

(25 pts) 1. **Safety First!**

It is 1944, and you are a physicist working on the Manhattan project, attempting to build an atomic bomb. In the process, you will be enriching a lot of uranium. The thing about this enriched uranium is, if too much of it gets close together, it will explode, making you very unhappy. Naturally, you will take precautions to make sure you don't get blown up, though ideally you'd like to do it as inexpensively as possible...

(a) The first approach is to take your storage facilities as fixed, and place the uranium in some subset of these locations that are sufficiently spaced out. We model the storage facility as an undirected graph $G = (V, E)$, where V is the set of locations at which we can store uranium. Each location can hold 1 unit of uranium. The edges have length $\ell(e)$, which induce a distance metric on the graph, where $d(u, v)$ is the cost of the minimum length path from u to v. You are given a parameter D. Putting uranium at any two locations u and v such that $d(u, v) < D$ will cause an explosion. You'd like to place as many units of uranium as possible down, subject to the constraint of not being blown up. Derive a polynomial time algorithm to maximize the number of units of uranium you can place (the output is the set of locations where you'd put non-zero amounts of uranium), or show that this problem is NP-hard.

(b) Another way to control the nuclear chain reaction is with control rods that stop the cascading neutrons. So you can imagine another undirected graph $G = (V, E)$ where you plan to place uranium at each and every location in V. This is safe if and only if you place control rods down at a set of locations $C \subseteq V$ such that each $v \in V$ is either in C, or has a neighbor in C. Your goal is the make the site safe while minimizing the size of C. Derive a polynomial time algorithm to minimize $|C|$, or show that this problem is NP-hard.

(c) A third way to deal with the uranium problem is to reinforce the walls with material that stops neutrons. Here you have an undirected graph $G = (V, E)$ with non-negative edge weights $w(e)$ and with a set $Y \subseteq V$ of locations where you plan to put the uranium. Each edge e represents a barrier to neutrons that can be reinforced any amount $\ell(e) \in \mathbb{R}_{\geq 0}$ at a cost of $w(e)\ell(e)$. You must reinforce the edges of the graph such that for any two vertices $u, v \in Y$, the shortest path from u to v under these lengths $\ell(e)$ (i.e., where you interpret edge e as having length $\ell(e)$) is at least one. Your goal is to find values of $\ell(e)$ to minimize the total cost you pay, subject to not being blown up. Derive a polynomial time algorithm to minimize the cost, or show that this problem is NP-hard.

Solution:

(a) This problem is NP-hard. Since it is a graph packing problem, we'll reduce Independent Set (IS) to it. Given IS instance (G, k) where $G = (V, E)$, construct a "Uranium Storage Problem Version One" (USP1) instance (G', w, D) as follows: $G' = G$, $w(e) = 1$ for all edges in G', and $D = 1.5$. Then output "yes, G has an independent set of size at least k" if and only if we can pack k or more units of uranium into G'.

We now prove correctness of the reduction. If we can place k units of uranium down at locations $S \subseteq V$ in G', then we know that S is an independent set, since $d(u, v) \geq 1.5$ for all distinct $u, v \in S$, and if there where an edge $(u, v) \in E$ then $d(u, v) \leq 1$. Similiarly, any independent set $T \subseteq V$ in G must have for all distinct $u, v \in T$, $d(u, v) \geq 2$ in G' under w. Thus the optimum solution to our USP1 instance is exactly the size of the largest independent set in G.

As an aside, the decision version of USP1 is in NP, and thus NP-complete, since we can verify a proposed solution easily in polynomial time.

(b) This problem is also NP-hard. Since it is a covering problem, we'll reduce another covering problem to it – Set Cover.

Start out with a Set Cover instance $(U, \{S_1, S_2, \ldots, S_m\}, k)$, where $S_i \subseteq U$ and $k \in \mathbb{N}$. Call our problem "Uranium Storage Problem Version Two" (USP2), which will be on graph $G = (V, E)$. First, construct the bipartite set-system graph: Add a vertex to V for each $u \in U$ and each S_i, and add edges (u, S_i) for each u and each S_i such that $u \in S_i$. Next add two more vertices to V, called r and r', as well as an edge (r, r') and edges (r', S_i) for each $1 \leq i \leq m$.

We claim that the Set Cover instance has a set cover of size at most k if and only if our USP2 instance has a solution of at most $k+1$. First, suppose there is a set cover $C \subseteq \{S_1, S_2, \ldots, S_m\}$. Then $C \cup \{r'\}$ is a solution to our USP2 instance, since every $u \in U$ is adjacent to some $S_i \in C$, each S_i is adjacent to r', and r is adjacent to r'.

Next, suppose there is an optimal USP2 solution X of cost $k' + 1$ for $k' \leq k$. WLOG, $r' \in X$ since X has to include at least one of $\{r, r'\}$ and it might as well include r' since r has degree one. Since it includes r', the S_i's are covered by r'. Thus we can assume WLOG that $X \cap U = \emptyset$. For suppose $u \in X \cap U$. Since all the S_i's are covered by r', we can obtain another solution by deleting u from X and adding any set S_i that contains u. This cannot increase the size of X. Thus, there exists an optimal USP2 solution X to our instance such that $X = \{r'\} \cup C$ for some $C \subseteq \{S_1, S_2, \ldots, S_m\}$. By the definition of USP2 solutions, C must be a set cover solution of size $k' \leq k$.

(As an aside, the decision version of USP2 is in NP, and thus NP-complete, since we can verify a proposed solution easily in polynomial time.)

(c) This problem, which we'll call (you guessed it) "Uranium Storage Problem Version Three" (USP3), can be solved in polynomial time via linear programming. Here's the LP, followed by an explaination.

$$\begin{array}{ll}
\min \sum_{e \in E} w(e)x(e) & \\
d(u, v) \geq 1 & \forall u, v \in Y \\
d(u, v) \leq d(u, w) + x(\{w, v\}) & \forall u, w, v \in V \text{ such that } \{w, v\} \in E \\
0 \leq x(e) \leq 1 & \forall e \in E
\end{array}$$

Here, $x(e)$ is the amount we are reinforcing edge e, and each $d(u, v)$ is a *variable* that intuitively serves as a lower bound on the distance between u and v under edge distances x. We define $d(v,v) := 0$ for all $v \in V$. Thus we have constraints like $d(u,v) \leq d(u,u) + x(\{u,v\})$ which simplify (after a macro replacement) to $d(u,v) \leq x(\{u,v\})$. In what follows, let $D_x(u,v)$ equal the distance between u and v under edge distances x.

We now show that this LP is a correct formulation of the problem.

Claim 1: For all LP solutions (x,d), for all $u,v \in V$, $d(u,v) \leq D_x(u,v)$.

Proof: Fix any solution (x,d) and fix any path P $x(P) := \sum_{e \in P} x(e)$. Let P be a path from u to v. Then we can prove $d(u,v) \leq x(P)$ by induction on the number of edges in P. That is, if P has one edge $\{u,v\}$, then $d(u,v) \leq x(\{u,v\})$ is a constraint in the LP, so we are done. For the induction hypothesis, we have that if $\{w,v\}$ is the last edge in P, and $Q := P \setminus \{w,v\}$, then $d(u,v) \leq d(u,w) + x(\{w,v\}) \leq \sum_{e \in Q} x(e) + x(\{w,v\}) = x(P)$ by the LP constraint and the induction hypothesis on Q. In particular this holds for the *shortest path* between each u and v under distances x.

Note that given x, we can always construct a solution (x,d) by setting $d(u,v) = D_x(u,v)$, and this has no effect on the cost. Actually, we can show that given x, (x,d) is an LP solution if and only if $d(u,v) \leq D_x(u,v)$.

Now, given a solution x to the original USP3 instance of cost W, we can construct an LP solution (x,d) of the same cost W using $d(u,v) = D_x(u,v)$. Conversely, given an LP solution (x,d), we know that $D_x(u,v) \geq d(u,v) \geq 1$ for all $u,v \in Y$, so x is a solution to the USP3 instance of the same cost. Thus solving the LP optimally solves the USP3 instance optimally.

(25 pts) 2. **What are Widgets, Anyways?**

You are in charge of the supply chain for Widgets-r-Us, where you have to find ways to transport materials from their storage warehouses to the factories, subject to capacity constraints on the transportation network. You model it as a directed graph $G = (V, E)$, where each directed edge $e \in E$ represents a one-way road. (Roads capable of carrying traffic in both directions can be thought of as two one-way roads.)

There are k "warehouse" vertices, and k kinds of materials, where the i^{th} material is originally located at the warehouse vertex $s_i \in V$, and the total amount of material i is D_i. Each road (directed edge) e has a capacity u_e, such that the total amount of material (of all kinds) sent over this road must be at most u_e. You may assume that all quantities given as input in this problem are non-negative integers.

You also have ℓ "factory" vertices. There are ℓ factories producing widgets (the j^{th} factory is located at the j^{th} factory vertex $f_j \in V$). The factory f_j has a request vector $\mathbf{r}_j = (r_{1j}, r_{2j}, \ldots, r_{kj})$, such that to produce one unit of widget, it requires r_{ij} amounts of material i for every $i \in \{1, \ldots, k\}$. [1]

(a) Write an LP to figure out how to transport the material from their warehouses to the factories (respecting the road capacity constraints), to maximize the total amount of widgets produced, subject to these constraints. (It is OK if you produce fractional amounts of widgets.)

(b) You find out that widgets produced at different factories sell for different amounts of money: the price per unit of widget produced at factory j is p_j. You want to maximize your revenue. Change the LP from the previous part to handle this.

(c) You are informed that two of the roads $e_1 = (u, v)$ and $e_2 = (v, u)$ are special: they represent the two directions of traffic over a bridge. For structural reasons, you have the balance requirement that the absolute value of the difference between the amount of material send in the two directions can be at most $\delta \in \mathbb{Z}_{\geq 0}$. How can you add such a constraint to your LP.

Solution: We'll have a variable w_j for each factory $j \in [\ell]$ which represents the amount of widgets produced at factory f_j. And a variable f^i_{uv} for each $i \in [k]$ and every edge (u, v) representing the amount of material i flowing along this edge. For ease of writing, we will introduce some auxiliary variables along the way. First, let us define the excess flow z^i_v for a node v and material i to be the amount of material i entering it minus that leaving it. These are new variables.

$$z^i_v = \sum_{u:(u,v)\in E} f^i_{uv} - \sum_{u:(v,u)\in E} f^i_{vu}. \qquad \forall i, v \tag{1}$$

[1] Nodes like s_i and f_j can have both incoming and outgoing arcs, and the route taking material i from the warehouse s_i to some factory f_j is allowed to pass through some other warehouse vertex $s_{i'}$ or some other factory vertex $f_{j'}$. You may assume that all the $k + \ell$ warehouse and factory vertices are distinct.

Then for material i, we have

$$z^i_{s_i} \geq -D_i \qquad \forall i \tag{2}$$
$$z^i_v = 0 \qquad \forall i, \forall v \notin \{s_i, f_1, f_2, \ldots, f_\ell\} \tag{3}$$
$$z^i_v \geq 0 \qquad \forall i, \forall v \in \{f_1, f_2, \ldots, f_\ell\} \tag{4}$$

Moreover, the amount of widgets produced is bounded above by the amount of excess flow of each material at the factories (scaled by the requests):

$$w_j \leq \frac{1}{r_{ij}} \cdot z^i_{f_j} \qquad \forall i, j \tag{5}$$

Finally,

$$w_j, f^i_{uv} \geq 0. \tag{6}$$

The objective function for part (a) is $\sum_j w_j$, and for part (b) is $\sum_j p_j w_j$.

For (c), we add in the two constraints:

$$-\delta \leq \sum_i f^i_{uv} - \sum_i f^i_{vu} \leq \delta.$$

(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 $v'_i := k \cdot \left\lfloor \frac{v_i}{k} \right\rfloor$. 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 $\varepsilon = 0.01$.

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 $\max\{v|s[n, v] \leq S\}$. We can calculate $s[m, v]$ by running a nested loop:

```
for $v = 1, 2, \ldots$ do:
  for $m = 0, 1, \ldots, n$ do:
    if $(m = 0)$ then $s[m, v] = \infty$,
    else if $(v_m \geq v)$ 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 $\sum_i k \cdot (\frac{v_i}{k} - 1) \leq \sum_i k\lfloor \frac{v_i}{k} \rfloor \leq \sum_i k\frac{v_i}{k}$.
It must be the case that $V' \leq V$, because otherwise the items in V' would be a valid solution to I with higher cost than V. Similarly, $V' \geq V - nk$, because the optimal solution to I has a value of at least $V - nk$ in I'.

(c) We first note that $V' \leq nv_{max}$, 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 $(v_m \geq vk)$ 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 $k = \frac{\varepsilon v_m ax}{n}$ 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 $V - nk = V - \varepsilon v_{max} \geq V(1 - \varepsilon)$, because every item fits into the knapsack so we can always get at least v_{max}.