15-451/651 Algorithms, Spring 2019

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

This HW has three regular problems and a programming assignment. All problems on the HWs are to be done *individually*, no collaboration is allowed. Solutions to the written problems should be submitted as a single PDF file using `gradescope`, with the answer to each problem starting on a new page.

The programming part is due on Wednesday April 10th. However, we will accept it without penalty until Saturday April 27th 11:59pm. Note you cannot add on the late days after April 27th, that is the very last day we will accept submissions.

(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.)

(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.

(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., $s_i \leq S$ 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 $(1 - \varepsilon)OPT(I)$.

(25 pts) 4. **Programming Problem: Ski Slopes**

You are building a new ride at Kennywood. This involves a collection of slides between various locations. In order to make the ride as safe as possible, you want to minimize the slope of the slides. But to make the ride interesting, the endpoints of some of the slides are fixed at certain heights.

We model this problem as an undirected graph $G = (V, E)$ where the edges are the slides, so that each edge $e \in E$ has a positive integer edge length $\ell(e)$, and a subset of vertices have a specified (possibly negative) integer height $h(v)$. Given an assignment of heights to all the vertices, the slope of an edge $e = (u, v)$ is defined as follows:

$$\text{slope}(e) = \frac{|h(u) - h(v)|}{\ell(e)}.$$

Your program should assign real valued heights to all the remaining vertices so as to minimize the maximum *slope* among all the edges.

Your program should be called `slope.c` or something analogous. We'll provide code for the simplex algorithm in some common languages, which you can use. This code has

the property that it has an implicit $x_i \geq 0$ non-negativity constraint for all *variables*, so you have to take that into account. You are allowed to use negative *constants* in your constraints and objective, of course.

Finally, 5 of the 25 points are for giving a short explanation of what your algorithm does (in the comments *above the code*, not interspersed in the code). We understand that the simplex algorithm takes exponential time in the worst-case, but you should argue that had you used a poly-time LP solver, you'd have got a correct poly-time algorithm for the above slope-assignment problem.

Input:

The first line of input contains n and m the number of vertices and edges of G. $1 \leq n \leq 1000$, $1 \leq m \leq 2000$. The following n lines each contain either the character "$*$", or an integer. The $*$ indicates that the hight is unspecified, and is to be computed by your program.

The following m lines each contain a pair of vertex numbers in the range $[0, n-1]$, and an integer representing the length of that edge. There are no multi-edges or self-loops. The length will be an integer in the range $[1, 10000]$. There will be no edges connecting a pair of vertices with known heights.

Output:

Output the minimum obtainable slope with six digits after the decimal point. Your answer should be within a relative or absolute error of at most 10^{-6} of the correct answer. The time limit is 10 seconds.

Examples:

```
3 2                                          slope = 5.000000
0
*
-100
0 1 1
1 2 19

6 5                                          slope = 3.500000
*
8
1
2
5
4
0 1 1
0 2 1
0 3 1
0 4 1
```

```
0 5 1
```