CSE421: Design and Analysis of Algorithms January 25, 2018

Homework 4

Shayan Oveis Gharan Due: Feb 1, 2018 at 5:00 PM

Please see https://courses.cs.washington.edu/courses/cse421/18wi/grading.html for general guidelines about Homework problems.

Most of the problems only require one or two key ideas for their solution. It will help you a lot to spell out these main ideas so that you can get most of the credit for a problem even if you err on the finer details. Please justify all answers.

P1) In class we discussed that we can solve the interval scheduling problem by sorting all jobs in the order of their finishing time. Show that if we sort the jobs in the order of the starting times, the greedy algorithm may not return the optimum. In other words, construct a set of jobs with their starting times and finishing times such that if we sort the jobs based on their starting times the greedy algorithm schedules a smaller number of jobs than the optimum.

P2) Given a sequence $d_1, \dots, d_n$ of integers design a polynomial time algorithm that construct a tree such that the degree of vertex i is d_i. If no such tree exists your algorithm must output "Impossible".

Hint: Show that for every sequence $d_1, \dots, d_n$ there exists a tree with this degree sequence if and only if $\sum_i d_i = 2(n-1)$ and for all i, we have $d_i \geq 1$. Also, you may have to argue that if the sum of n integers is less than $2n$ then one of them is at most 1.

P3) In this problem we want to use DFS to solve the following problem: Given a tree T with n vertices and a set of pairs $(u_1, v_1), \dots, (u_k, v_k)$ we want to design an $O(n + k \log n)$ time algorithm to find the length of the unique path from u_i to v_i in T for all i.

a) Show how to modify the code for recursive depth-first search of undirected graphs to assign to each node v a number, dfsnum[v], indicating the sequence number for when it was first discovered by depth-first search. This is exactly the number that we printed next to each discovered vertex in DFS-Tree slides.

b) To obtain the length of the path from u_i to v_i first we find the lowest common ancestor of u_i and v_i in T and then we add up the distance of u_i and v_i to the lowest common ancestor. In this part we construct a data structure that helps us find the lowest common ancestor. Construct a **sorted** array st[.] that at anytime (when running the DFS) contains the dfsnum[v] for every node v which is discovered but not yet fully explored. In other words, st[] contains all nodes in the stack where their DFS call is still running. Modify the code of DFS to construct this array.

c) Fix a pair u_i, v_i and suppose we first discover u_i and then v_i. Show that the lowest common ancestor of u_i, v_i is the largest j such that st[j] $\leq$ dfsnum[u_i] at the time that we call dfs(v_i).

d) Fix a pair u_i, v_i and suppose we first discover u_i and then v_i. Observe that when we call dfs(v_i) we know dfsnum[u_i]. Use part (c) to design an algorithm that returns the dfsnum of the lowest common ancestor of u_i, v_i.

e) Design an $O(n + k \log n)$ time algorithm to find the distance of all u_i, v_i in the tree T. You would also receive full credit in this part if your algorithm runs in $O((n + k) \log n)$.

P4) You are given a graph G with n vertices and m edges, and a minimum spanning tree T of the graph. Suppose one of the edge weights $w(e)$ of the graph is updated. Give an algorithm that runs in time $O(n+m)$ to test if T still remains the minimum spanning tree of the graph. Your algorithm should output "yes" if T is still the MST and "no" otherwise. You may assume that all edge weights are distinct both before and after the update.

P5) **Extra Credit:** One of the first things you learn in calculus is how to minimize a differentiable function like $y = ax^2 + bx + c$, where $a > 0$. The minimum spanning tree problem, on the other hand, is a minimization problem of a very different flavor: there are now just a finite number of possibilities for how the minimum might be achieved – rather than a continuum of possibilities – and we are interested in how to perform the computation without having to exhaust this (huge) finite number of possibilities.

One can ask what happens when these two minimization issues are brought together, and the following question is an example of this. Suppose we have a connected graph $G = (V, E)$. Each edge e now has a time-varying edge cost given by a function $f_e : \mathbb{R} \to \mathbb{R}$. Thus, at time t, it has cost $f_e(t)$. We'll assume that all these functions are positive over their entire range. Observe that the set of edges constituting the minimum spanning tree of G may change over time. Also, of course, the cost of the minimum spanning tree of G becomes a function of the time t; we'll denote this function $c_G(t)$. A natural problem then becomes: find a value of t at which $c_G(t)$ is minimized.

Suppose each function f_e is a polynomial of degree 2: $f_e(t) = a_e t^2 + b_e t + c_e$, where $a_e > 0$. Give an algorithm that takes the graph G and the values $\{(a_e, b_e, c_e) : e \in E\}$, and returns a value of the time t at which the minimum spanning tree has minimum cost. Your algorithm should run in time polynomial in the number of nodes and edges of the graph G. You may assume that arithmetic operations on the numbers $\{(a_e, b_e, c_e)\}$ can be done in constant time per operation.