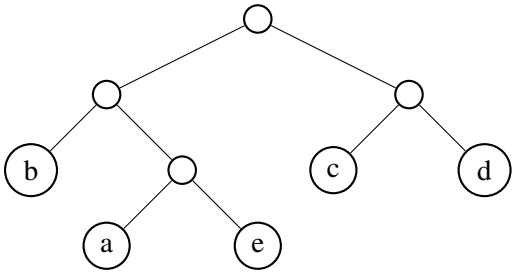
CS 170 Efficient Algorithms and Intractable Problems
Spring 2017 Prasad Raghavendra and Sanjam Garg

Midterm 2 Review

March 18, 2017

1. Short Questions

1. Give a Huffman encoding tree for the alphabet $\{a,b,c,d,e\}$ with frequencies $f_a = 0.15$, $f_b = 0.32$, $f_c = 0.19$, $f_d = 0.24$, $f_e = .1$.

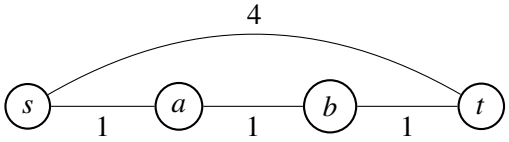


2. Consider an undirected graph $G = (V,E)$ with nonnegative edge weights $w_e \geq 0$. Suppose that you have computed a minimum spanning tree of G, and that you have also computed shortest paths to all nodes from a particular node $s \in V$.

   Now suppose each edge weight is increased by 1: the new weights are $w'_e = w_e + 1$.

   (a) Assuming G has a unique minimum spanning tree T, is T still an MST of the new graph? Prove or provide a counterexample.
   Solution: The MST does not change because every spanning tree has exactly $|V|-1$ edges, so the weight of each one is increased by exactly $|V|-1$. Thus, the original MST still has a smaller weight than every other spanning tree, so it is still the MST.

   (b) Can the shortest path between two vertices change? Prove that it can't or provide an example in which it does.
   Solution: The shortest $s-t$ path is $s \to a \to b \to t$, but when we add 1 to each edge, the shortest $s-t$ path is $s \to t$.



3. Under a Huffman encoding of n symbols with frequencies $f_1, f_2, \ldots, f_n$, what is the longest a codeword could possibly be? Give an example set of frequencies that would produce this case.
   Solution: $n-1$. For $i = 1,\ldots,n-1$, $f_i = \frac{1}{2^i}$. $f_n = \frac{1}{2^{n-1}}$

4. We have an MST $T = (V,E')$ of a graph $G = (V,E)$. If we increase the weight of some $e \in E'$, give an algorithm to determine the new MST T'.
   Solution: Let $e = (u,v)$. Consider the cut defined by all the vertices reachable from u in T without using e and all the vertices reachable from v in T without using e. To get T', replace e with the edge with minimum cost across this cut in E.

2. Lexicographically smallest subsequence

You are given a string S and a length k. Give a dynamic programming algorithm to find the smallest (by lexicographical ordering) subsequence of length exactly k. Note that a subsequence must retain characters in the same ordering as in S, but need not be contiguous. For example: in the word "rocket", the smallest subsequence of length 3 is "cet". For the purposes of this question you can assume concatenating two strings together takes constant time.

Solution. Use a dynamic programming approach:

Let $C[i]$ be the smallest subsequence of length k over $S[1 \dots i]$. Then

$$C[i] = \min\{C[i-1], \min_{0 \le j < k} \Big\{ C[i-1][0 \dots j-1] + C[i-1][j+1 \dots k] + S[i] \quad \}$$

Interpretation: at each step of the recurrence, you can either use the same smallest subsequence as before, or include this new character $S[i]$ to replace one of the previous ones. The index j is picking out which letter to remove from the previously calculated smallest subsequence of length k over $S[1 \dots i-1]$.

Base case: $C[k] = S[1 \dots k]$. Runtime: each step of the algorithm takes time $O(k)$ and there are $n-k$ total sub-problems to solve, therefore the total runtime will be $O(nk)$. Note that an $O(n^2k)$ formulation is also possible by defining the sub-problems over both i and k, but is less efficient.

3. Longest common increasing subsequence

You are given two sequences $a = a_1, a_2, \dots, a_n$, and $b = b_1, b_2, \dots, b_m$, and you want to find a subsequence such that it is a subsequence of both a and b, is strictly increasing, and is as long as possible.

For example, the longest common increasing subsequence of $(1,2,4,2,1,6,8)$ and $(2,1,4,2,4,4,2,6)$ is $(1,2,4,6)$. We can verify that there is no longer subsequence that is both strictly increasing and a subsequence both sequences.

Devise an efficient algorithm to solve return the length of the longest common increasing subsequence (algorithms can optionally find and return the subsequence as well).

Input: Two sequences of integers a, b.

Output: A single integer, denoting the length of the longest common increasing subsequence.

The runtime of your algorithm should be $O(nm(n+m))$.

Solution. First, let's remap our integers so they are between 1 and $n+m$ inclusive. Let $G(i,j,k)$ be the longest common increasing subsequence of the sequences $a_1, \dots, a_i$ and $b_1, \dots, b_j$ and that the largest element of the common subsequence is at most k. So, we have the recurrence

$$G(i,j,k) = \max(G(i-1,j,k), G(i,j-1,k), G(i,j,k-1), G(i-1,j-1,k-1) + equals(a_i, b_j, k))$$

where $equals(x,y,z)$ returns 1 if and only if $x = y = z$.

The proof works as follows, either a_i is not in our LCIS, b_j is not in our LCIS, we have a LCIS where the largest element is at most $k-1$, or $a_i = b_j = k$ is part of our LCIS.

The running time is $O(nm(n+m))$, since there are $nm(n+m)$ states in our dp, and each state takes constant time to evaluate.

4. Disease prevention

You are in charge of preventing the spread of a disease in a faraway nation called Treeland. The cities in Treeland are labeled from 1 to n. In addition, some cities are considered neighbors, and you are given that

as a list of pairs. Interestingly, the cities are connected in such a way that they form the nodes of a connected undirected, unweighted tree. You'd like to open some disease prevention clinics in some of the cities in such a way that all cities become "safe". A city is "safe" if either (a) it has a disease prevention clinic, or (b) **all** of its neighbors have a disease prevention clinic. The cost of opening a disease prevention clinic in city i is equal to c_i. Each c_i will be positive. Given n, the number of cities, the list of pairs of cities that are considered neighbors, and $c_1, c_2, \ldots, c_n$, devise an efficient algorithm to determine the minimum cost you need to spend in order to make every city safe. Your algorithm may optionally return the set of cities in which we should open a clinic.

Input: An integer n, and a list of $n-1$ pairs of integers, where each pair describes two neighboring cities. In addition, you are given a length n array of positive integers $c_1, \ldots, c_n$.

Output: A single integer, denoting the minimum cost to make every city safe.

The runtime of your algorithm should be at most $O(n)$.

Solution:

First, notice that the cities in which we choose to open disease prevention clinics must form a *vertex cover* – that is, for every edge $(u,v) \in E$, we must build a clinic in either u or v. With that in mind, we can follow a solution that modifies the vertex cover on trees problem from discussion.

Arbitrarily root the tree, and make each undirected edge directed from parent to child. Let's define our subproblem to be $C[u]$, the minimum cost of making completely safe all cities in the subtree rooted at u. Then our recurrence is

$$C[u] = \min \begin{cases} c_u + \sum_{v:(u,v)\in E} C[v] & \text{use vertex } u \\ \sum_{v:(u,v)\in E} (c_v + \sum_{w:(v,w)\in E} C[w]) & \text{use all children of vertex } u \end{cases}$$

We can compute these values from bottom up, from children to the root. The base case is $C[u] = 0$ for all leaves u. Our final answer is $C[r]$ where r is our root. There are n subproblems each of which take constant time to solve, so our overall runtime is $O(n)$.

5. Water Supply via Linear Programming

There are four major cities and three water reservoirs in California.

- Reservoir i holds G_i gallons of water, while city j needs D_j gallons of water.
- It costs p_{ij} dollars per gallon of water supplied from reservoir i to city j.
- The capacity of the piping from reservoir i and city j can handle at most c_{ij} gallons.
- In view of fairness, no city must get more than $1/3$rd of all its water demand from any single reservoir.

Write a linear program to determine how to supply the water from the reservoirs to the cities, at the lowest cost.

1. What are the variables of the linear program, and what do they indicate?

   Solution:
   We define f_{ij} to be the amount of water, in gallons, supplied from reservoir i to city j.

   Common Mistakes:

- Many students included the provided parameters (e.g. G_i, p_{ij}) or the number of cities/reservoirs as variables. These are not variables of the linear program, but parameters; the linear program cannot set these variables as part of the optimization process (e.g. if p_{ij} was a variable, then the optimal solution is to set it to 0).
- Some students defined multiple new groups of variables, some of which were redundant (e.g. in addition to defining f_{ij} as above, but also g_i as the total flow out of reservoir i). This wasn't incorrect, but this often led to errors in part c, as described below.

2. What is the objective function being maximized/minimized?

Solution:
We wish to minimize $\sum_{ij} p_{ij} \cdot f_{ij}$, the cost of supplying the water.

3. What are the constraints of your linear program? (No need to list every constraint, but list one of each type and explain how the rest are generated)

Solution:
We have five groups of constraints. First, the constraints:

$$\forall i : \sum_j f_{ij} \leq G_i$$

$$\forall j : \sum_i f_{ij} \geq D_j$$

$$\forall i, j : f_{ij} \leq c_{ij}$$

$$\forall i, j : f_{ij} \leq D_j/3$$

$$\forall i, j : f_{ij} \geq 0$$

The first group corresponds to the limit on how much water each reservoir can support.
The second group corresponds to the requirement of how much water each city must get.
The third group corresponds to the maximum capacity of the piping.
The fourth group corresponds to the fairness requirement.
The fifth group corresponds to a sanity check on flow; we cannot send negative amounts of flow.

Common Mistakes:

- Many students did not include the fifth constraint.
- Conversely, some students imposed nonnegativity constraints on the input parameters. This does sanity check the input, but was not needed in the solution.
- Some students simply neglected one or more of the constraints, particularly the constraints for D and G.
- It is important to emphasize that, in general, linear programs cannot have strict inequalities as constraints (e.g. $>$ and $<$ cannot appear in a linear program). None of the constraints in the problem needed a strict inequality.
- If in part (a), the student defined multiple variables, it was important to explicitly relate these variables as constraints. For example, if f_{ij} and g_i were defined as in the common mistake above, then a constraint $\sum_i f_{ij} = g_i$ was needed; otherwise, the two variables would not be related in the LP.