CS 577: Introduction to Algorithms — Homework 5

Out: 04/17/12 — **Due: 04/24/12**

Ground Rules

Same as for homework 1.

Problems

1. **(5 points)** In a tribute to Dr. Seuss, here is a question based on his story "Yertle the turtle". You don't need to know the story to solve the question.

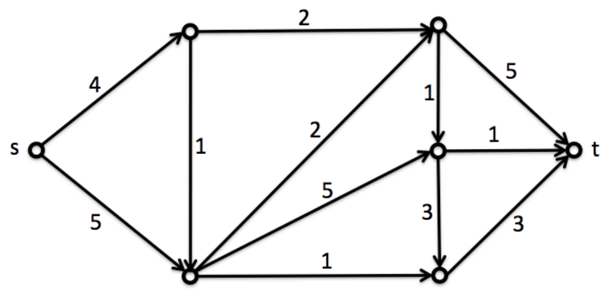
   Mack, in an effort to avoid being cracked, has enlisted your advice as to the order in which turtles should be dispatched to form Yertle's throne. Each of the turtles ordered by Yertle has a different weight and strength. Your task is to build the largest stack of turtles possible such that for every turtle in the stack, its strength is no less than the total weight that it carries above it.

   You are given the weight and the strength of each turtle. The strength is the turtle's overall carrying capacity, including its own weight. That is, a turtle weighing 300 units with a strength of 1000 units could carry other turtles weighing a total of 700 units on its back.

   Your algorithm should output the maximum number of turtles that can be stacked without exceeding the strength of any one. It should run in time $O(n^2)$, where n denotes the number of turtles given. You will be given partial credit for an algorithm that runs in time polynomial in n and the weights or strengths of the turtles.

2. **(2+3 points)** For this question, you are given a directed network $G = (V, E)$ with capacities c_e on edges and a source-sink pair (s, t). In part (b) you are also given the max s-t flow in the graph, f^*. An edge in the network is called a *bottleneck* edge if increasing its capacity by one unit increases the max flow in the network.

   (a) For the network below determine the max s-t flow, f^*, a min s-t cut (note that this may not be unique), and the residual graph G_{f^*}. Also identify all of the bottleneck edges in the graph.

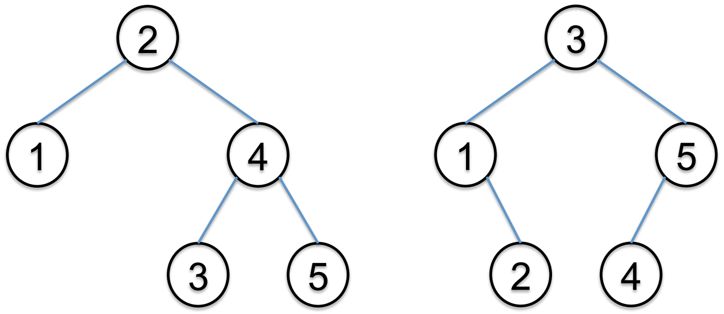


   (b) Develop an algorithm for determining all the bottleneck edges in the given graph. Your algorithm is allowed to use the max flow f^* and should run in linear time.

3. **(5 points)** One way to search efficiently is to use binary search trees. A binary search tree is a binary tree where each node is labeled with one of the elements and the labels satisfy the following property — the label of every node is larger than the label of every node in its left subtree, and smaller than the label of every node in its right subtree. Two examples of binary search trees over the elements $\{1, 2, 3, 4, 5\}$ are given below.

   Given the frequencies $f_1, \cdots, f_n$ of searching for elements $1, \cdots, n$, the average cost of search is equal to $\sum_i f_i \ell_i$ where ℓ_i is the "depth" of element i, or the length of the path from the root to the node labeled i. Binary search trees are therefore just like Huffman codes with two differences—elements can be assigned to internal nodes, and the assignment must satisfy the ordering described above.

Give an algorithm for constructing the optimal binary search tree (minimizing the average cost of search) given n elements $1, \cdots, n$ and their frequencies $f_1, \cdots, f_n$. Your algorithm should run in time polynomial in n.



4. **(Extra credit.)** Give an $O(n^2)$ time algorithm for the optimal binary tree problem using the following fact (that you do not need to prove): Let $r(i, j)$ denote the index of the root of the optimal tree over elements $i, i+1, \cdots, j$; Then $r(i, j-1) \leq r(i, j) \leq r(i+1, j)$ for all $i < j$.