Sample Solution to Midterm

Problem 1. (10 points) Colorful spanning tree

Give a polynomial-time algorithm that takes as input an undirected unweighted connected graph G with each edge colored either green or yellow, and an integer k, and determines whether there exists a spanning tree with *exactly* k green edges. Justify the correctness of the algorithm, and analyze its worst-case running time.

Answer: Consider spanning trees T_g and T_y of G containing the minimum number of green and yellow edges, respectively. Both of these exist since G is connected and can be found by setting weights of green and yellow edges to be 0 and 1 (1 and 0, respectively) and computing the MSTs. We claim that there exists a spanning tree with exactly k edges if and only if the number of green edges in T_g is at most k and that in T_y is at least k. For the if direction, we use the claim proved in Problem Set 2 to "go" from T_g to T_y, swapping one edge at a time; since the number of green edges can increase by at most 1 in each swap, it follows that at some instance, we will obtain a spanning tree with exactly k green edges. The only if direction trivially holds.

The above argument requires two MST calculations and implies an $O(E \lg V)$ algorithm. A faster $O(V + E)$ algorithm can be obtained as follows. To calculate T_g, remove all green edges, and find connected components: contract each connected component to one node, and find any spanning tree of the contracted graph, all of whose edges are green. One can do a similar process for T_y.

If one needs to compute the actual spanning tree, then one can start with any spanning tree T and proceed as follows. If T has more than k green edges, then go through each green edge one by one until the number of green edges left is k, or we have processed all green edges – Consider the cut obtained when the edge is removed; if there is any yellow edge that crosses the cut, then replace the green edge by the yellow edge. If the loop terminates with exactly k green edges left, then we have the desired tree; otherwise, no such tree exists.

Problem 2. (10 points) Locating Mirror Sites in a Network

A simplified model for a computer network with a set $\mathcal{M}$ of n machines is a hierarchical structure given by a rooted tree T. The tree has n leaves, one leaf for each of the n machines in the network. Each node v in the tree has a label $\ell(v)$. The label of every leaf node is 0, and all the labels satisfy the following condition: the label of a node is at most the label of its parent. The communication cost $d(X, Y)$ between two machines X and Y, referred to as the *distance* between X and Y, is given by the label of the least common ancestor of X and Y in T. For a given number $p \leq n$, you are asked to determine a placement of p mirror sites in the given network. For each machine $X \in \mathcal{M}$ you are given a weight $w(X)$ that represents the frequency with which the machines accesses the site. The goal is to determine a placement such that the total weighted distance of the machines to the nearest mirror site according to the placement is minimized.

Formally, a placement specifies a set P of p machines where the mirror sites are located. Let $n_P(X)$

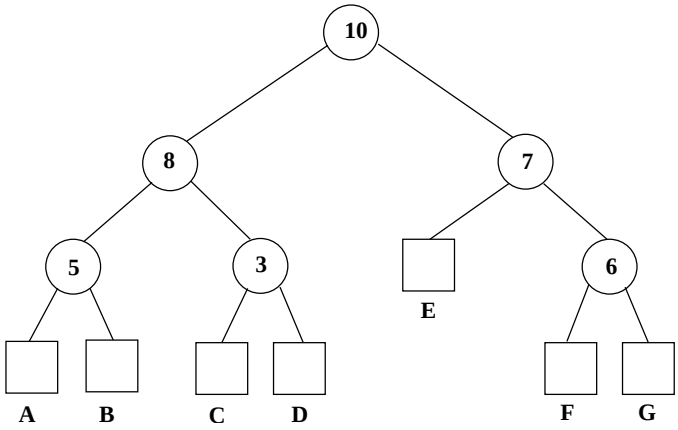

Figure 1: Illustration of the tree hierarchy. The machines are the (rectangular) leaves. The label of each non-leaf node is shown; these labels which are used in calculating the distances. For instance, $d(C, E)$ is 10 while $d(E, G)$ is 7.

denote the machine in P that is nearest to X; that is, if P is nonempty,

$$n_P(X) = \arg\min_{Y \in P} d(X, Y).$$

The cost of placement P is given by

$$\sum_{X \in \mathcal{M}} w(X) d(X, n_P(X)).$$

As an example, consider the tree in Figure 1. Suppose we are given $w(A) = w(B) = w(C) = w(D) = 1$ and $w(E) = w(F) = w(G) = 2$. Then the cost of placement $\{A, E, F\}$ will be $1 \cdot 0 + 1 \cdot 5 + 1 \cdot 8 + 1 \cdot 8 + 2 \cdot 0 + 2 \cdot 0 + 2 \cdot 6 = 33$.

Design an efficient algorithm for finding a minimum-cost placement of p mirror sites for an n-machine network, whose hierarchical structure is given as a *full binary tree*; that is, every non-leaf node in the tree has exactly two children. (The tree in Figure 1 is a full binary tree.) Your algorithm must run in time polynomial in n and p. Justify the correctness of your algorithm and analyze its worst-case running time.

Answer: We present two algorithms. One a dynamic programing algorithm with running time $O(np^2)$, and the other a greedy algorithm with running time $O(np)$.

- **Dynamic programming:** We will compute a function M over $T \times \{0, 1, \ldots, p\}$ such that $M(u, j)$ returns an optimal placement of at most j mirror sites in the tree rooted at node u. Let $C(P)$ denote the cost of placement P. Let $W(u)$ denote the total weight of all of the nodes in the tree rooted at u.

  We have the following recurrence:

  - If $j = 0$, then $M(u, j) = \emptyset$.

– If $j > 0$ and u is a leaf node, then $M(u, j) = \{u\}$.
– We consider the case $j \geq 0$ and u a non-leaf node. Let x and y be the two children of u. $M(u, j)$ equals the argmin of three terms: $\min_{1 \leq i \leq j-1} C(M(x, i)) + C(M(y, j - i))$, $C(M(x, j)) + W(y)\ell(u)$, and $C(M(y, j)) + W(x)\ell(u)$.

The computation time for the W-vector is $O(n)$. The computation time for each $M(u, j)$ is $O(j)$. Thus, the computation time for $M(u, j)$, over all j, is $O(p^2)$. The total computation time for all of the $M(\cdot, \cdot)$ values is $O(np^2)$.

The correctness of the algorithm follows from an optimal substructure property that can be shown easily. We leave the proof as an exercise for the reader.

- **Greedy algorithm:** A naive greedy approach is to select the node that is farthest away (in a weighted sense) from its nearest mirror site. It can be shown that this approach will not work, since the other nodes near this selected node may have little weight, thus contributing little benefit to including the mirror at the selected node. So a better strategy is to place a mirror site at the node that results in the maximum benefit.

  Algorithm: Let $cost(P)$ denote the cost of a placement P. Given a current placement P, define the benefit of adding a machine X to P as $cost(P) - cost(P \cup \{X\})$. The greedy algorithm iteratively adds the node to the placement that maximizes the benefit, until the placement has p nodes. (The cost of the empty placement is set to the product $W(r) \cdot Max$, where r is the root of the tree and Max is an arbitrary number greater than $\ell(r)$.)

  The proof of the correctness of the greedy algorithm is somewhat tricky.

  Proof of correctness. Let k be an arbitrary integer that is at least 0 and less than p. Let P_k be a placement of k mirror sites that is a subset of an optimal solution of p mirror sites. Let A denote the machine that maximizes the benefit $cost(P_k) - cost(P_k \cup \{A\})$. Then $P_k \cup \{A\}$ is a subset of an optimal solution of p mirror sites.

  Let P^* denote an optimal solution of p mirror sites such that P_k is a subset of P^*. (By our assumption P^* exists.) If A is also in P^*, then there is nothing to prove. Otherwise, let B denote a machine in $P^* - P_k$ that is nearest to A. Let $T_{A,B}$ denote the smallest subtree that contains both A and B. We compare the costs of P^* and $P^* + \{A\} - \{B\}$. We know that

$$\begin{aligned} cost(P^*) &= cost(P_k) - (cost(P_k + B) - cost(P_k)) - (cost(P_k + B) - cost(P^*)) \quad (1) \\ cost(P^* + A - B) &= cost(P_k) - (cost(P_k + A) - cost(P_k)) \\ &\quad -(cost(P_k + A) - cost(P^* + A - B)) \quad (2) \end{aligned}$$

  We now argue that $cost(P_k+B)-cost(P^*)$ is the same as $cost(P_k+A)-cost(P^*+A-B)$. To see this, we note (i) for any X in $T_{A,B}$, $d(X, n_{P_k+B}(X)) = d(X, n_{P^*}(X))$ and $d(X, n_{P_k+A}(X)) = d(X, n_{P^*+A-B}(X))$; and (ii) for any X not in $T_{A,B}$, we have $d(X, n_{P_k+B}(X)) = d(X, n_{P_k+A}(X))$ and $d(X, n_{P^*}(X)) = d(X, n_{P^*+A-B}(X))$. It thus follows that for every X, we have:

$$d(X, n_{P_k+B}(X)) - d(X, n_{P^*}(X)) = d(X, n_{P_k+A}(X)) - d(X, n_{P^*+A-B}(X)),$$

  which implies the desired claim

$$cost(P_k + B) - cost(P^*) = cost(P_k + A) - cost(P^* + A - B)$$

  Putting this equality into Equations 1 and 2 and noting that

$$cost(P_k + B) - cost(P_k) \quad \geq \quad cost(P_k + A) - cost(P_k),$$

we obtain that $cost(P^* + A - B) \leq cost(P^*)$, thus completing the desired claim. **End of proof**.

Running time. Each iteration of the greedy algorithm requires the computation of the cost of at most n placements. The cost of each placement can be computed in $O(n)$ time by taking advantage of the fact that the placement includes just one more mirror site than the one computed in the previous iteration (so we can reuse the computation in the previous iteration). Since there are p iterations, the running time is $O(n^2p)$.

We can, in fact, improve the running time to $O(np)$ by computing, for each additional mirror site, its optimal location in $O(n)$ time. The idea is to compute this using two passes through the tree. For each node v, we compute two costs: optimal cost $I(v)$ incurred if an additional mirror site is placed in the tree rooted at v, and cost $N(v)$ incurred if an additional mirror site was not placed in the tree rooted at v but in the subtree rooted at the sibling of v. For a node v, $I(v)$ is simply $\min\{I(v_l) + N(v_r), I(v_r) + N(v_l)\}$, where v_l and v_r are the left and right children of v; if v is a leaf, then $I(v)$ is zero. The cost $N(v)$ is the current cost if there already is a site in the subtree rooted at v, and $w(v) \cdot \ell(p)$, otherwise, where p is the parent of v. If v is the root, then $N(v)$ need not be defined.

From the above definitions, $I(v)$ and $N(v)$ can be computed using a bottom-up pass through the tree. Following this calculation, the optimal location for an additional site can be computed by a top-down pass through the tree; if we have decided to place the new site in the subtree rooted at v, and v children are v_l and v_r, then we place it in the subtree rooted at v_l if $I(v_l) + N(v_r) \leq I(v_r) + N(v_l)$, and in the subtree rooted at v_r otherwise. It thus follows that the greedy algorithm can be implemented in time $O(np)$.

Problem 3 (10 points) Selecting online advertisements

Major online portals like Google and Yahoo have considerable information about the individual users based on their past interactions. This allows them to post targeted advertisements to the users. Suppose a set U of n users, labeled 1 through n, visit the portal on a particular day. The portal has a set A of m ads, labeled 1 through m, to choose from. The analysis of the users has revealed k different groups (from a marketing standpoint), the ith group consisting of subset S_i of users from U. A user may be part of several groups; i.e., a user may be an element of several different S_i's. The jth ad is targeted to a subset $G_j \subseteq \{1, \ldots, k\}$ of the groups.

The portal needs to decide whether there exists a way of assigning advertisements to users such that the following conditions hold: (a) each user is shown exactly one ad; (b) ad j is shown to user i only if i is in a group k in G_j; (c) the number of times the ad j is shown is at least r_j, where r_j is a given integer.

Give a polynomial-time algorithm that takes the above input – U, A, the sets S_i's, the groups G_j's, the r_j's – and determines whether the portal can assign an ad to each user so that the above three conditions are satisfied, and if so, then returns such an assignment.

Answer: We set up a network flow problem as follows. We have a node for each user, for each ad, for each group, and a source node s, and sink node t. Thus, the set of nodes is $U \cup A \cup \{S_i : 1 \leq i \leq k\} \cup \{s, t\}$. The edges and their capacities are as follows. There is an edge from s to every user u in U, with capacity 1. There is an edge from user u to group S_i if $u \in S_i$ with capacity ∞. There is an edge from group S_i to ad j if j can be shown to users in S_i; i.e., if $i \in G_j$ with capacity ∞. Finally, there is an edge from ad j to t with capacity r_j.

We compute maximum flow and determine whether value of flow equals $\sum_j r_j$. If it is, then the flow gives an assigment of ads to users satisfying all the constraints, except for possibly not covering all users. We can then assign ads to remaining users arbitrarily, while ensuring the constraint that user is in one of the appropriate groups.

We now consider the flow in the other direction. Suppose there is a way to assign the ads so that all constraints are satisfied. Consider the flow associated with this assignment. The only capacity constraints that could possibly be violated are the ones on the edges joining ads to the sink. Consider one such violation: suppose edge from ad j to t has a flow of $f_j > r_j$ on it; we compute $f_j - r_j$ flows, each of value 1 from the users to ad j to t, and remove them. Once we have addressed all violations, we have a max flow of value $\sum_j r_j$ in the original network.

Clearly, the algorithm is polynomial time.

Problem 4. (10 points) Privacy of survey data

This was the year of the Census. The results of a survey such as the Census are often disclosed in aggregate form to ensure the confidentiality of the information. Revealing nothing other than the aggregates also has a downside since that may provide too little information.

Suppose a survey has produced a $m \times n$ table D, in which each entry d_{ij} is a nonnegative integer. Let $r(i)$ be the sum of the elements in the ith row of D, and let $c(j)$ denote the sum of he elements in the jth column of D. The surveyor would like to disclose all the m row sums and the n column sums. Furthermore, the surveyor wants to disclose a subset, say S, of the mn matrix elements, and yet suppress the remaining matrix elements to ensure the confidentiality of the supressed information. Unless care is exercised, the surveyor may permit someone to deduce the exact value of one or more of the suppressed elements. We say that an entry d_{ij} of the matrix is *unprotected*, if it is possible to deduce the exact value of d_{ij} from the row and column sums and the elements of S.

Given m, n, the row and column sums, and the elements of S, our problem is to identify all the unprotected elements of the matrix and their values.

(a) (10 points) Using the maximum flow problem, give a polynomial-time algorithm for the above problem. Justify the correctness of your algorithm. You need not worry about the complexity of your algorithm, as long as it is polynomial-time. For instance, you may make multiple (polynomially many) calls to maximum flow subroutines, if needed.

Answer: We construct a network flow instance G as follows. The node set is as follows: a node R_i for each row i, a node C_j for each column j, a source node s and a sink t. For each row i, let $r'(i)$ denote $r(i) - \sum_{(i,j)\in S} d_{ij}$. For each column j, let $c'(j)$ denote $c(j) - \sum_{(i,j)\in S} d_{ij}$. We have an edge from s to each row node R_i with capacity $r'(i)$, an edge from each column node C_j to t with capacity $c'(j)$, and an edge from R_i to C_j for each $(i,j) \notin S$ with capacity ∞.

We now give an algorithm to determine whether a particular element (i,j) is protected. We compute a maximum flow f of G. Note that the flow value should equal $\sum_i r'(i)$ or $\sum_j c'(j)$. Let f_{ij} be the flow on the edge from R_i to C_j. We next compute a max flow on an instance G', which is identical to G except that we set the capacity of edge (R_i, C_j) to $f_{ij} - 1$. If we find that the maxflow in G' has value equal to $|f|$, then (i,j) is protected – since d_{ij} can take value f_{ij} as well as a value less than f_{ij}. Otherwise, we compute a maxflow on an instance G'',

which is identical to G except that the capacity of the edges (s, R_i) and (C_j, t) are reduced to $r'(i) - f_{ij} - 1$ and $c'(j) - f_{ij} - 1$, respectively. This checks whether d_{ij} can be greater than f_{ij}. If there is a maxflow in G'' of value at least $\sum_i r'(i) - f_{ij} - 1$, then again (i, j) is protected. Otherwise, (i, j) is not protected.

We repeat the above for all pairs (i, j) in S.

A number of you proposed alternative approaches in which the set of linear equations involving the variables is analyzed to see whether multiple solutions are possible. Those approaches could be made to work but one has to be careful that they take into account the fact that the d_{ij}s are all supposed to be nonnegative integers. The network flow approach always ensures that this condition is satisfied.

(b) (6 points) If you are unable to solve (a), then give a polynomial-time algorithm for the above problem using linear programming. Justify the correctness of your algorithm. Again, you need not worry about the complexity of your algorithm, as long as it is polynomial-time.