Sample Solution to Problem Set 4

1. (10 points) Rate-Optimal Spanning Tree

You are working at a telecommunications company that has access to a communication network G connecting a set V of nodes with a set E of undirected edges (links). Each link e in E has an associated bandwidth b_e. For any path P in G that connects two nodes, say u and v, we define the *rate* $b(P)$ of P to be the minimum, over all edges e in P, of b_e. For any pair u and v of vertices in G, the *best rate achievable between u and v in G* is the maximum of $b(P)$ over all paths P from u to v in G.

The era of cost-cutting has hit your firm as well. One idea proposed for trimming costs is to not use the entire network G, but instead use a *spanning tree* T of G.

(a) Show that there exists a spanning tree T of G that has the following (seemingly incredible) property: for *every pair* u and v of vertices in G, the best rate achievable between u and v in T is equal to the best rate achievable between u and v in G.

(b) Give an efficient algorithm to compute a spanning tree that satisfies the property stated in (a).

Answer:

(a) In this problem, we first give a proposal to build this spanning tree by greedy algorithm, and prove it is the spanning tree satisfying the requirement.

1) Initially we have a set of vertices called V_{new} including only one starting point, and E_{new} denotes as the empty edge set.
2) Each time we select the maximum edge $e(u, v)$ with one node u in V_{new} and one node v not in V_{new}, (if there are multiple edges with the same weight, any of them may be picked). Then add v into V_{new}, and add $e(u, v)$ into E_{new}.
3) Repeat until V_{new} is V, so V_{new} and E_{new} are the output spanning tree.

We claim the spanning tree built by the above is the required spanning tree in the problem. The prove for correctness is as follows:

1) Let the Case $|V_{new}| = 1$; $V_{new} = x$ and $d(x) = 0$. The claim holds.
2) Assume it holds when $|V_{new}| = k$ for some value of $k >= 1$. We now grow V_{new} to size $k + 1$ by adding the node v.
   Let $e(u, v)$ be the final edge on our V_{new} to V path. Call this path P_v. Suppose we have another path to v that is shorter. It would have crossed from V_{new} to $V - V_{new}$ at some point, where the boundary edge would be from u' to v'. However, based on our greedy step in the algorithm, the shorter path u to v was picked instead, so it is contradiction to have a shorter path going through u' and v' to v.

Thus our greedy algorithm holds for any case, and it will produce the required spanning tree.

(b) The algorithm for this solution is similar to Prim's Algorithm in which you grow the spanning tree (until complete) with a node v that maximizes the attachment cost.

The algorithm can be described as follows:

$V_{new} = \{x\}$, where x is an arbitrary node (starting point) from V.
$E_{new} = \{\}$
$d(s) = 0$
while $V_{new} \neq V$ **do**
  Select a node v not in V_{new} with at least one edge from V_{new} for which $d'(v) = max(l_e)$ for $e(u, v)$, where u in V_{new}
  Add v to V_{new}, add $e(u, v)$ to E_{new}
end while
return V_{new} and E_{new}

Starting from any node x would result in the same tree being spanned, so spanning tree T always contains the best achievable rate paths for u to v in G.

2. (20 points) Problem 5.21.

Answer:

(a) Denote the vertices set of this cycle as C, and the maximum weight edge as $e(u, v)$.

1) Suppose we are in one step of the track of building the spanning tree. At this step, C is cut to two parts: V_1 and V_2, in which V_1 includes u, and V_2 includes v.
2) Since C represents a cycle, there are at least two edges that connect V_1 and V_2, in which at least one edge is not e, suppose we call it as e'.
3) From the above, we know the weight of e' is no more than e. We apply the *cutproperty* to this case, there exists a minimum spanning tree which doesn't include edge e.

(b) According to the algorithm, we first sort the edges by the weights of them.

1) Each time if there still has a cycle in the graph, we will pick the heaviest weight edge e in the cycle. By the problem (a), we will know there exists a minimum spanning tree which doesn't include edge e.
2) So we just remove this heaviest edge e, we are sure about that the remaining graph still includes a minimum spanning tree.
3) Repeat the above steps, until we have no cycle in the remaining graph, which is the minimum spanning tree.

(c) Denote the edge e as $e(u, v)$. Each time when we want to check whether there is a cycle containing $e(u, v)$, we can do it as follows:

1) First we unconnect $e(u, v)$, in which we don't really remove the edge from the graph, we just mark it as unconnected.
2) Then we do a DFS from the vertex u. If it can reach the vertex v, we return $ture$, else return $false$.

In this algorithm, the unconnect step will only take constant time, and the overall running time will be linear to the input size of the graph.

(d) The algorithm described in the problem (b) includes the following steps:

1) Sort the edges by the weights of them. It will take $O(|E| \log |E|)$ by selecting proper sorting algorithm.
2) For each edge e, when we want to check whether there is a cycle containing e, it takes $O(|E|)$ time according to the problem (c). Thus to check every edge in the graph, we have to take $O(|E|^2)$ time.

So the overall running time of the above algorithm will be $O(|E| \log |E|) + O(|E|^2) = O(|E|^2)$

2. Problem 5.20 of online text

This problem is Problem 5.21 of the published text. Some of the students worked on this problem.

Perfect matching for tree: In any matching of a tree, every leaf has to be matched to its parent. Therefore, a tree has a perfect matching only if every leaf has no sibling. So our algorithm is to repeatedly (a) find/maintain leaves; (b) if any leaf has a sibling, then return no; (c) else, match each leaf to its a parent, delete both leaf and parent from graph, marking new leaves as appropriate.

Minimum feedback edge set of an undirected graph: We assume that the given graph is connected; otherwise, we calculate connected components and apply the following algorithm separately to each component. Let $G = (V, E)$ be the given connected graph. If S is a minimum feedback edge set of G, then $E - S$ forms a maximum spanning tree. Thus, all we have to do is find the maximum spanning tree (which is similar to finding minimum spanning tree) and return the edges not in the maximum spanning tree.

3. (10 points) Problem 5.29.

Answer: Suppose we are given an alphabet of size n. It is tempting to consider the following generalization of Huffman's algorithm. If $n \leq 2$, then have a root r with its children the nodes corresponding to the two characters. If $n \geq 3$, then select the 3 characters with minimum frequency (say a, b, and c) and make these characters siblings with a single parent, introduce a new corresponding to the parent assigning it with the frequency being the sum of the frequencies of a, b, and c, and then recurse.

There is a problem with the above approach, though. Notice that the above algorithm may produce a tree in which every internal node has 3 children except for the root which has 2 children; this is because n will be 2 only at the end of the computation. An optimal ternary code, however, should not have the root with only two children (for $n > 2$). For instance, consider the example with 4 characters a, b, c, and d with frequencies 1, 2, 3, and 4, respectively. The above algorithm will have a, b, and c as siblings and their parent being a sibling of d. The overall cost will be $2 * (1 + 2 + 3) + 4 = 16$. The optimal solution, however is to have a and b as siblings and their parent as sibling with c and d to give a cost of $2 * (1 + 2) + 3 + 4 = 13$.

So how do we generalize Huffman encoding? If we are guaranteed that every internal node has 3 children, then we can apply the above greedy choice. But the preceding condition may not always hold, as we saw in the case $n = 4$. However, we can see that in an optimal tree there can be only

one internal node with fewer than 3 children; furthermore, such an internal node should have two children and should be an internal node with largest depth. Why?

First, any internal node u with only child v can be removed and v made the child of the parent of u, hus decreasing the cost of the tree. Now consider the case when there are two internal nodes u and v each having two children. Let u_1 and u_2 be the children of u and v_1 and v_2 be that of v. Without loss of generality, suppose that the depth of u is smaller than the depth of v. We can remove v and make v_1 the child of u and v_2 the child of the parent of v, thus decreasing the cost of the tree, again a contradiction. We leave it as an exercise to the reader to argue that if an internal node does have fewer than 3 children it must be an internal node with largest depth.

We now know that an optimal prefix-free ternary code can be one of two kinds: (i) every internal node has exactly 3 children; or (ii) there exists one internal node with 2 children, all others have 3 children, and the node with 2 children has largest depth. Given a character set of size n, can we determine which of the two types the optimal tree T is? Somewhat surprisingly, yes. Let k be the number of internal nodes in T. If T is of type (i), then the number of edges in the tree, which is $k + n - 1$, also equals $3k$. We thus get $2k = n - 1$. Since k and n are integers, it follows that n is odd. On the other hand, if T is of type (ii), then the number of edges in the tree, which is $k + n - 1$, also equals $3k - 1$. We thus get $n = 2k$. Hence, we have proved that the optimal tree is of type (i) if n is odd; otherwise, it is of type (ii).

We thus have the following algorithm. If n is 1, there is nothing to do. If $n \geq 2$ and odd, then we select the 3 lowest frequency characters, make them siblings, add a parent character with frequency of their sum and repeat. Note that when we repeat, the new value of n is two less than before, implying that the new value is still odd. Similarly, if $n \geq 2$ and even, then we select the 2 lowest frequency characters, make them siblings, add a parent character with frequency of their sum and repeat. Now when we repeat, the new value of n is odd. And from now on, the algorithm will repeatedly select the 3 lowest frequency characters until there is exactly one character left.

Optimal substructure property: The proof for the optimal substructure property is almost identical to that for the binary case; the only difference is that we are working with ternary trees and the proof we went through in class is working with binary trees.

Greedy property: If n is odd, then we select the 3 smallest frequency characters and make them siblings. We have already argued that when n is odd, the optimal tree is of type (i). Given this, we can invoke a swapping argument similar to the binary case to claim that the 3 smallest frequency characters have to be siblings. When n is even, we know that the optimal tree is of type (ii). Therefore, the internal node with two children needs to be of largest depth. We invoke a swapping argument similar to the binary case to claim that the 2 smallest frequency characters have to be siblings, and their parent must be the internal node with two children.

3. Problem 5.28 of online text

This problem is Problem 5.29 of the published text. Some of the students worked on this problem.

We went through an argument in class. Consider a binary tree in which an edge to a left child is labeled 0, and an edge to a right child is labeled 1. A codeword corresponds to a unique path in this tree; we place the element with the codeword as the endpoint of this path; since this codeword is not a prefix of any other codeword, we can make this element a leaf. We now have a binary tree with each element of the alphabet as a leaf.