College of Computer & Information Science	CS5800
Northeastern University	MS Algorithms
Fall 2018	8 November 2018

Sample Solution to Problem Set 5

Problem 1. (5 points) Three-character Huffman codes

Consider the problem of Huffman codes where we use three characters from $\{0, 1, 2\}$ in our code, as opposed to the bits 0 and 1. Modify the Huffman encoding algorithm to determine a minimum-length compression of any sequence of characters from an alphabet A of size n, with the ith letter of the alphabet having frequency $f[i]$. Your algorithm should encode each character with a variable-length codeword over the values $\{0, 1, 2\}$ such that no codeword is a prefix of another codeword and so as to obtain the maximum possible compression. Prove that your algorithm is correct. Analyze the worst-case running time of your algorithm.

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 three-ary 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 three-ary 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.

Greedy choice: 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.

The remainder of the proof – that once the 3 or 2 smallest frequency characters are combined, the remaining problem is to compute an optimal solution for the instance with the combined characters – is almost identical to that for the binary case the only difference is that we are working with three-ary trees and the proof we went through in class is working with binary trees.

Problem 2. (5 points) Matching Widgets and Gadgets

You are given a set W of n widgets and a set G of n gadgets. Each widget w has a weight $W(v)$ and each gadget g has a weight $W(g)$. You would like to match each widget w in W to a unique gadget g in G so as to minimize the sum of the absolute values of the weight differences of the matched pairs. That is, you would like to find a perfect matching M between W and G that minimizes

$$\sum_{(w,g)\in M} |W(w) - W(g)|.$$

Design an efficient greedy algorithm to solve the given problem. Prove that your algorithm is correct. Analyze the worst-case running time of your algorithm.

Answer:

Claim 1 *Let w be a widget with maximum weight and let g be a gadget with maximum weight. There is an optimal pairing in which w is paired with g.*

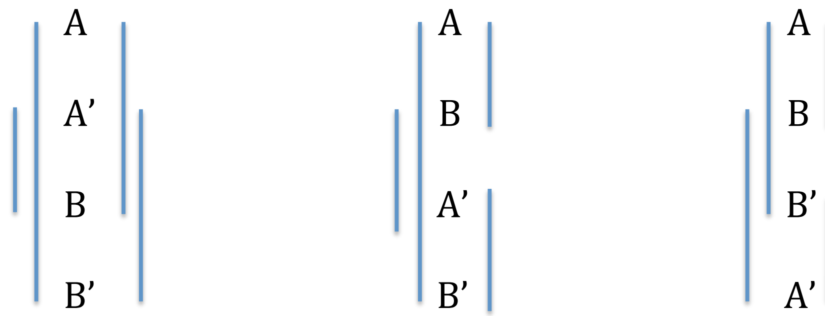
Proof: The proof is by contradiction. Consider an optimal pairing. Suppose w is paired with g' with lower weight than g, and g with w' with lower weight than w. Let the weights of w, w', g, and g' be $W(w)$, $W(w')$, $W(g)$, and $W(g')$. We show that

$$|W(w) - W(g)| + |W(w') - W(g')| \leq |W(w) - W(g')| + |W(w') - W(B)|,$$

contradicting the assumption that the given pairing is optimal. We consider three cases.

1. $W(w) \geq W(w') \geq W(g) \geq W(g')$: In this case, both sides are equal to $W(w) - W(w') + 2(W(w') - W(g)) + W(g) - W(g')$.
2. $W(w) \geq W(g) \geq W(w') \geq W(g')$. In this case, the right-hand side exceeds the left-hand side by $2(W(g) - W(w'))$.
3. $W(w) \geq W(g) \geq W(g') \geq W(w')$. In this case, the right-hand side exceeds the left-hand side by $2(W(g) - W(g'))$.

In the figure below, A, B, A', and B' refer to $W(w)$, $W(g)$, $W(w')$, and $W(g')$, respectively. The values are organized top to bottom in decreasing order. The length of a line segment is the absolute



value of the relevant difference.

■

Problem 3. (3 + 2 = 5 points) Uniqueness of MSTs when all weights are distinct

(a) Suppose T_1 and T_2 are distinct minimum spanning trees for graph G. Let (u, v) be the lightest edge (smallest weight edge) among all edges that are in T_1 and but not in T_2. Let (x, y) be any edge that is in T_2 and not in T_1. Show that $w(x, y) \geq w(u, v)$.

Answer: Suppose we add edge (x, y) to T_1. This creates a cycle C. By the MST property, it follows that the weight of every edge in C is at most $w(x, y)$. Furthermore, at least one edge in C is in T_1 and not in T_2, because otherwise T_2 has a cycle. Let this edge be (a, b). Thus, it follows that $w(x, y) \geq w(a, b)$. Since (u, v) is the lightest edge that is in T_1 and not in T_2, we have $w(u, v) \leq w(a, b)$. This implies that $w(x, y) \geq w(u, v)$, thus proving the desired claim.

(b) Using part (a), prove that if the weights on the edges of a connected, undirected graph are distinct, then there is a unique minimum spanning tree.

Answer: Let, if possible, there be two distinct minimum spanning trees T_1 and T_2. We will derive a contradiction. Let (u, v) be the smallest weight edge in T_1 that is not in T_2, and let (u', v') be the smallest weight edge in T_2 that is not in T_1. Since all the edges weights are distinct, we have either $w(u, v) < w(u', v')$ or $w(u', v') < w(u, v)$. Without loss of generality, assume the former. Now suppose we add the edge (u, v) to T_2. This forms a cycle, say C, in T_2. Cycle C contains at least one edge that is in T_2 and not in T_1, because otherwise T_2 has a cycle. Let this edge be (x, y). By the definition of (u', v'), we have $w(u', v') \geq w(x, y)$. But $w(u', v') > w(u, v)$, implying that $w(x, y) > w(u, v)$, violating the MST property of T_2. This yields a contradiction.

Problem 4. (5 points) Leaf-Constrained Spanning Tree

Design an algorithm, which takes as input a connected undirected graph $G = (V, E)$, a weight function $w : E \to Z^+$, and a subset U of V, and returns minimum-weight spanning tree of G satisfying the property that every vertex in U is a leaf in T. If no such spanning tree exists, then your algorithm must indicate so. Analyze the worst-case running time of your algorithm.

(*Note:* The desired spanning tree may not be a minimum spanning tree of G. The spanning tree your algorithm returns must satisfy the desired property and have the minumum weight among all spanning trees satisfying the desired property.)

Answer: Let T be a spanning tree of G that satisfies the property that every vertex in U is a leaf in T, and has minimum weight among all spanning trees satisfying the property. We establish two properties about T.

First, since every vertex in U is a leaf in T, if we remove all the vertices in U from G, then $T' = T \setminus U$ is a spanning tree of $G' = G \setminus U$. Indeed, it is a *minimum spanning tree* of $G \setminus U$ since otherwise, we can replace $T \setminus U$ by an MST of $G \setminus U$ and add the lone edges for each $u \in U$ back in to obtain a tree with the desired property and yet lower weight than that of T, leading to a contradiction.

Second, for each vertex u in U, let $e(u)$ denote a minimum-weight edge in the set $\{(u, v) : v \in V \setminus U\}$. We argue that there exists a choice of T which includes $e(u)$, since otherwise we can replace the unique edge adjacent to $u \in U$ in T and obtain a new spanning tree of no more weight than T.

These two properties suggest the following algorithm.

1. Compute graph G' by removing all vertices in U and their incident edges.
2. If G' is not connected, then return "No such spanning tree exists".
3. Compute an MST T' of G' using Prim's or Kruskal's algorithm.
4. For each $u \in U$, compute $e(u)$ as the minimum-weight edge between u and any vertex in $V \setminus U$.
5. Return $T \leftarrow T' \cup \{e(u) : u \in U\}$.

We now analyze the running time of the algorithm. Step 1 takes linear time $\Theta(n + m)$. Step 2 takes linear time using DFS. Step 3 takes $\Theta(m \log n)$ time. Step 4 takes linear time. Finally, Step 5 takes linear time. So the total time is $\Theta(m \log n)$. Here m and n are the number of edges and vertices respectively in G.