CSCE 411-200, Fall 2016
Homework 4 Solutions

Problem 1: Exercise 17.4-3 (p. 471): Suppose that instead of contracting a table by halving its size when its load factor drops below 1/4, we contract it by multiplying its size by 2/3 when its load factor drops below 1/3. Using the potential function

$$\Phi(T) = |2 \cdot T.num - T.size|$$

show that the amortized cost of a TABLE-DELETE that uses this strategy is bounded above by a constant.

Solution: This exercise shows that by changing the rule for contractions, a simpler potential function is possible!

Since deletes never cause expansions, we only have to consider the case when a contraction doesn't occur and the case when a contraction occurs. I'll use n_i for $T_i.num$ and s_i for $T_i.size$.

First suppose there is no contraction (i.e., the table is at least 1/3 full both before and after the delete). The actual cost is just 1 (remove the element).

$$\begin{aligned}
m_i &= c_i + \Phi(T_i) - \Phi(T_{i-1}) \\
&= 1 + |2 \cdot n_i - s_i| - |2 \cdot n_{i-1} - s_{i-1}| \\
&= 1 + |2(n_{i-1} - 1) - s_{i-1}| - |2 \cdot n_{i-1} - s_{i-1}| && \text{since } num \text{ reduces by 1 and } size \text{ stays same} \\
&\le 1 + |(2(n_{i-1} - 1) - s_{i-1}) - (2 \cdot n_{i-1} - s_{i-1})| && \text{since } |a| - |b| \le |a - b| \\
&= 1 + 2 \\
&= 3.
\end{aligned}$$

Now suppose there is a contraction (i.e., the table is at least 1/3 full before the delete, but removing one element causes it to drop below 1/3 full). The actual cost is n_{i-1} as one element is removed and the remaining elements are copied. (This argument is due to Xing Zhao and assumes that the old table is exactly 1/3 full and we ignore floors and ceilings; a similar but slightly more involved argument works in the more general case.)

$$\begin{aligned}
m_i &= c_i + \Phi(T_i) - \Phi(T_{i-1}) \\
&= n_{i-1} + |2 \cdot n_i - s_i| - |2 \cdot n_{i-1} - s_{i-1}| \\
&= n_{i-1} + |2 \cdot (n_{i-1} - 1) - s_i| - |2 \cdot n_{i-1} - s_{i-1}| && \text{since } num \text{ decreases by 1} \\
&= n_{i-1} + |2 \cdot n_{i-1} - 2 - s_i| - |2 \cdot n_{i-1} - s_{i-1}| \\
&= \tfrac{1}{3} \cdot s_{i-1} + \left|\tfrac{2}{3} \cdot s_{i-1} - 2 - \tfrac{2}{3} \cdot s_{i-1}\right| - \left|\tfrac{2}{3} \cdot s_{i-1} - s_{i-1}\right| && \text{since } n_{i-1} = \tfrac{1}{3} \cdot s_{i-1} \text{ and } s_i = \tfrac{2}{3} \cdot s_{i-1} \\
&= \tfrac{1}{3} \cdot s_{i-1} + |-2| - \left|-\tfrac{1}{3} \cdot s_{i-1}\right| \\
&= \tfrac{1}{3} \cdot s_{i-1} + 2 - \tfrac{1}{3} \cdot s_{i-1} \\
&= 2.
\end{aligned}$$

Problem 2: Problem 17-5 (pp. 476–477): Competitive analysis of self-organizing lists with move-to-front.

Solution: List has length n; access sequence is $\sigma = \langle \sigma_1, \ldots, \sigma_m \rangle$.

(a) Show if heuristic H does not know the access sequence in advance, then the worst-case cost for H on an access sequence σ is $C_H(\sigma) = \Omega(mn)$.

Let L be the initial list and L_i^* be the state of the list after the i-th access using heuristic H. Suppose σ_1 is the last element in L; then the cost of σ_1 is $\Omega(n)$, since the entire list must be traversed to find the element, not counting the work of any transpositions that H might do. Suppose σ_2 is the last element in L_1^*; similarly, the cost of σ_2 is $\Omega(n)$. And so on for all m elements of σ for a total of $\Omega(mn)$.

———-

(b) Show that if σ_i accesses element x in list L using the move-to-front heuristic, then cost of σ_i is $c_i = 2\cdot$ $\text{rank}_L(x) - 1$.

A cost of $\text{rank}_L(x)$ is incurred to find x since you have to search through the list from the beginning until reaching x; then $\text{rank}_L(x) - 1$ transpositions are done to move x to the front. In other words, $c_i = 2\cdot$ $\text{rank}_L(x) - 1$.

———-

(c) Show that $c_i^* = \text{rank}_{L_{i-1}^*}(x) + t_i^*$, where c_i^* is the cost of access σ_i using the arbitrary heuristic H and t_i^* is the number of transpositions that H does during this access.

A cost of $\text{rank}_{L_{i-1}^*}(x)$ is incurred to find x since you have to search through the list from the beginning until reaching x; then t_i^* transpositions are done.

————

Define $\Phi(L_i)$ to be 2 times the number of inversions in L_i w.r.t. L_i^*.

(d) Show that a transposition either increases the potential by 2 or decreases the potential by 2.

Suppose the transposition involves elements y and z.

Case 1: (y, z) is not an inversion before the transposition. Then after the transposition, there is exactly one more inversion, and the potential increases by 2.

Case 2: (y, z) is an inversion before the transposition. Then after the transposition, there is exactly one fewer inversion, and the potential decreases by 2.

————

Suppose access σ_i wants to find x. Define:
$A =$ elements that precede x in both L_{i-1} and L_{i-1}^*
$B =$ elements that precede x in L_{i-1} only
$C =$ elements that precede x in L_{i-1}^* only
$D =$ elements that precede x in neither.

(e) Show that $\text{rank}_{L_{i-1}}(x) = |A| + |B| + 1$.

A and B consist of exactly those elements that precede x in L_{i-1}, and they are disjoint. Thus x comes after those $|A| + |B|$ elements and no others.

Show that $\text{rank}_{L_{i-1}^*}(x) = |A| + |C| + 1$.

A and C consist of exactly those elements that precede x in L_{i-1}^*, and they are disjoint. Thus x comes after those $|A| + |C|$ elements and no others.

————

(f) Show that access σ_i causes a change in potential of $\Phi(L_i) - \Phi(L_{i-1}) \le 2(|A| - |B| + t_i^*)$.

$$\begin{aligned}\Phi(L_i) - \Phi(L_{i-1}) &= 2 \cdot (\text{number of inversions in } L_i) - 2 \cdot (\text{number of inversions in } L_{i-1}) \\ &= 2 \cdot (\text{change in number of inversions})\end{aligned}$$

So we want to show that the change in the number of inversions is at most $|A| - |B| + t_i^*$.

First, consider the inversions in L_{i-1} w.r.t. L_{i-1}^*. By the definition of A, each element in A does not form an inversion with x. By the definition of B, each element in B does form an inversion with x.

When going from L_{i-1} to L_i, the only change is that x now appears at the beginning of the list. As an intermediate step, consider the inversions in L_i w.r.t. L_{i-1}^*. Each element a in A now forms an inversion with x, since x now precedes each such element a in L_i, but a precedes x in L_{i-1}^*. Each element in B now stops forming an inversion with x, since x now precedes each such element b in L_i and x precedes b in L_{i-1}^*. Thus the number of inversions in L_i w.r.t. L_{i-1}^* differs from the number of inversions in L_{i-1} w.r.t. L_{i-1}^* by $|A| - |B|$.

Finally, consider the inversions in L_i w.r.t. L_i^*. In going from L_{i-1}^* to L_i^*, t_i^* transpositions are performed, each of which causes at most one additional inversion w.r.t. L_i. Thus the final change in the number of inversions is at most $|A| - |B| + t*_i$.

(g) Show that the amortized cost m_i of access σ_i is bounded from above by $4 \cdot c_i^*$.

$$\begin{aligned}
m_i &= c_i + \Phi(L_i) - \Phi(L_{i-1}) \\
&= 2 \cdot \text{rank}_{L_{i-1}} - 1 + \Phi(L_i) - \Phi(L_{i-1}) \text{ by part (b)} \\
&\leq 2 \cdot \text{rank}_{L_{i-1}} - 1 + 2(|A| - |B| + t_i^*) \text{ by part (f)} \\
&= 2(|A| + |B| + 1) - 1 + 2(|A| - |B| + t_i^*) \text{ by part (e)} \\
&= 4|A| + 1 + 2 \cdot t_i^* \\
&\leq 4|A| + 4|C| + 4 + 4 \cdot t_i^* \text{ since } |C| \text{ and } t_i^* \text{ are nonnegative} \\
&= 4(|A| + |C| + 1 + t_i^*) \\
&= 4(\text{rank}_{L_{i-1}^*}(x) + t_i^*) \text{ by part (e)} \\
&= 4 \cdot c_i^* \text{ by part (c)}
\end{aligned}$$

(h) Conclude that the cost of move-to-front is at most 4 times the cost of any other heuristic (for the same sequence of accesses when starting with the same list).

The cost of the entire sequence of accesses is at most $\sum_{i=1}^{m} m_i$ since the potential function is valid. By part (g), this is at most

$$\sum_{i=1}^{m} 4 \cdot c_i^* = 4 \sum_{i=1}^{m} c_i^*$$

which is 4 times the cost of the entire sequence of accesses using the arbitrarily chosen other heuristic.

Problem 3: Problem 21-3 (pp. 584–585): Tarjan's off-line least-common ancestors algorithm.
Solution:

(a) Show Line 10 executes exactly once for each $\{u, v\} \in P$.

The algorithm is basically depth-first search, so all nodes are visited (and colored black) exactly once. The only possibility for line 10 to be executed is during LCA(u) or LCA(v). WLOG, suppose u is visited before v. During LCA(u), line 9 will be false (v is not black) and so line 10 will not be executed. But during LCA(v), line 10 will be true (u is black) and so line 10 will be executed.

(b) Show when LCA(u) is called, the number of sets equals the depth of u in T.

Lemma 1: When LCA(u) finishes, there is a set that consists exactly of u and all its descendants.

Proof: By strong induction on the *height* of u. The basis is when u is a leaf. When LCA(u) begins, a set is made for u. Since u is a leaf, it has no children, no unions are done, and at the end of LCA(u), there is a singleton set containing just u. Assume that the lemma is true for all nodes at height less than h (i.e., strong induction) and show the lemma is true for all nodes at height h. Consider LCA(u) where u has height $h > 0$. Thus all of u's children are at height less than h. By the inductive hypothesis, when each child of u finishes its LCA call, there is a set consisting exactly of that child and all its descendants. Since each of these sets is then unioned with u's set, the lemma is true when LCA(u) finishes.

End of Proof of Lemma 1.

Consider node u whose path from the root is $v_0, v_1, \ldots, v_{k-1}, v_k = u$. When LCA$(u)$ is called, there is exactly one set for each v_i on the path and this set contains v_i and all its descendants that are "to the right" of

v_{i+1} (assuming the DFS considers children in left-to-right order). This is due to Lemma 1 and the fact that nodes to the right of this path have not yet been visited and thus no MAKE-SET has been called for them.

(c) Prove that LCA correctly prints the least common ancestor of u and v for each pair $\{u, v\} \in P$.

Part (a) shows that line 10 will be executed during LCA(v) (letting v be the element of the pair that is visited second). We must show that at that time, FIND-SET$(v).ancestor$ equals u.

Recall that FIND-SET(v) returns an element of v's set that we consider the "representative" (or name) of the set. The "ancestor" variable field of the representative is assigned in lines 2 and 6. Note that this variable is always assigned to be the parent after unioning with the set for the child. Thus FIND-SET$(v).ancestor$ always holds the node that is at the lowest depth among all the nodes in the set.

For convenience, let w be FIND-SET$(v).ancestor$. Since u is "to the right" of v in the tree and has already been visited, and by the argument above that w is at the lowest depth, it follows that w is also an ancestor of u. However, this is the first time (greatest depth) at which this holds, so w is the LCA of u and v.

(d) Time complexity using tree-based implementation of disjoint sets data structure with weighted union and path compression.

Let n be the number of nodes in T. Since T is a tree, it has $n - 1$ edges. There are n MAKE-SET operations, one for each node. There are $n-1$ UNION operations from all the executions of line 5. There are n FIND-SET operations from all the executions of line 2, $n-1$ FIND-SET operations from all the executions of line 6, and $|P|$ FIND-SET operations from all the executions of line 10. Thus the time complexity from the disjoint sets data structure operations is $O((n + |P|)\log^* n)$.

Most of the remaining work takes time proportional to n in total (the loop bookkeeping, the recursive call bookkeeping, line 7). However, line 9 (checking if the other node in a pair is black) is done $|P|$ times in total. Furthermore, the time for line 8 (finding every node v that is in a pair with u) will take time depending on what data structure is used to represent the pairs. For instance, if the pairs are just in a list, then we'd have to traverse the entire list, taking $|P|$ time *in each recursive call.* Note that $|P|$ could be as large as $\Theta(n^2)$. So the total running time is: $O((n + |P|)\log^* n + n|P|)$.

Problem 4: Exercise 16.3-7 (p. 436): Generalize Huffman's algorithm to ternary codewords (i.e., codewords using the symbols 0, 1, and 2), and prove that it yields optimal ternary codes. *Hints for the proof of optimality:* Assume number of characters is at least 3 and is odd. Also, you may use without proof the fact that in an optimal tree for a ternary code, every non-leaf node has exactly three children.
Solution:

```
Input:  set C of characters, each with a frequency
Output:  labeled ternary tree corresponding to a code for C

make a tree node for each c in C
insert(Q,c) for each c in C
while Q has at least 3 elements do
   allocate a new node v
   v.left = x = extract_min(Q); label 0
   v.center = y = extract_min(Q); label 1
   v.right = z = extract_min(Q); label 2
   v.freq = x.freq + y.freq + z.freq
   insert(Q,v)
endwhile
```

Now show that the output of this algorithm is an optimal ternary code.

Use induction on n, the number of characters in C. Assume n is odd. (So induction will go from $n-2$ to n, in order to keep the number of characters odd.)

Basis: $n = 3$. The algorithm gives code 0 to one character, code 1 to another character, and code 2 to the third character, depending on the relative order of occurrence of the characters. Clearly a 1-symbol code is the smallest possible.

Induction: Suppose the algorithm is optimal for $n-2$ characters. We must show it is optimal for n characters.

Let C be a set of n characters, with occurrences given by f. Let $T_H(C)$ be the tree constructed by the algorithm. Consider characters x, y and z with the third fewest, second fewest and fewest occurrences. By the way the algorithm works, x, y and z are leaves and are siblings in $T_H(C)$.

Claim 1: In an optimal tree for a ternary code, every non-leaf node has exactly three children. We are allowed to use this claim without having to proof it.

Claim 2: In some optimal tree (i.e., tree corresponding to an optimal code) for C, x, y and z are also leaves and siblings.

Proof of Claim:

1. z must be at the greatest depth in any optimal tree. Otherwise swapping z with another character at greater depth would produce a better code.
2. By Claim 1, z has two siblings.
3. z's siblings are also leaves, because z has greatest depth.
4. y is also at greatest depth. Otherwise swapping y and one of z's siblings would produce a better code.
5. y has two siblings that are also leaves (same argument as for z).
6. If y and z are not siblings, then swap y with one of z's siblings and get another optimal code.
7. Repeat the same argument for x as for y to make x be a sibling of y and z.

End of Proof of Claim 2.

Let $T_{opt}(C)$ be an optimal tree for C in which x, y and z are leaves and siblings.

Let $C' = C - \{x, y, z\} \cup \{w\}$ where w is a new character with $f(w) = f(x) + f(y) + f(z)$. That is, remove x, y and z, and replace them with a new character whose number of occurrences is the sum of those of x, y and z.

Note that C' has $n-2$ characters in it, so we can apply the inductive hypothesis to C': i.e., the algorithm produces an optimal code for C'.

Let $T_H(C')$ be the tree produced by the algorithm on C'. By the way the algorithm works, $T_H(C')$ is the same as $T_H(C)$ (the algorithm's tree for C) except that the leaves for x, y and z, together with their parent, are replaced by the single node u, which is a leaf.

Let T^* be the result of doing the same replacement (of x, y, z and their parent with w) to $T_{opt}(C)$.

T^* represents a code for C'. We don't know if it's an optimal code, but its cost is certainly no smaller than that of an optimal code.

We want to show that $cost(T_H(C)) \leq cost(T_{opt}(C))$, i.e., that the cost of the tree produced by the algorithm for C is at least as small as the optimal cost for C.

$$\begin{aligned} cost(T_H(C)) &= cost(T_H(C')) + f(x) + f(y) + f(z) \text{ see (a) below} \\ &\leq cost(T^*) + f(x) + f(y) + f(z) \text{ by the INDUCTIVE HYPOTHESIS } T_H(C') \text{ is optimal} \\ &= cost(T_{opt}(C)) \text{ see (b) below} \end{aligned}$$

So if we can prove (a) and (b), we have shown that the cost of the tree T made by the algorithm for C is at most the cost of the optimal tree, and thus T is optimal also.

Now prove (a). Let d be the cost (i.e., depth) of x, y and z in T. Then the cost (i.e., depth) of w is $d-1$ in T'. So we have:

$$\begin{aligned} cost(T_H(C')) &= cost(T_H(C)) - cost(x) - cost(y) - cost(z) + cost(w) \\ &= cost(T_H(C)) - f(x)\cdot d - f(y)\cdot d - f(z)\cdot d + (f(x)+f(y)+f(z))\cdot(d-1) \\ &= cost(T_H(C)) - f(x) - f(y) - f(z). \end{aligned}$$

Now rearrange to get $cost(T_H(C)) = cost(T_H(C')) + f(x) + f(y) + f(z)$.

Now prove (b). We want to show that

$$cost(T_{opt}(C)) = cost(T^*) + f(x) + f(y) + f(z).$$

This is done the same way as (a).

Problem 5: Exercise 23.2-8 (pp. 637–638): Professor Borden proposes a new divide-and-conquer algorithm for computing minimum spanning trees. Given $G = (V, E)$, partition V into V_1 and V_2 such that $|V_1|$ and $|V_2|$ differ by at most 1. Let E_i be the set of edges incident only on V_i for $i = 1, 2$. Recursively find the MSTs of the subgraphs $G_i = (V_i, E_i)$ for $i = 1, 2$. Select minimum-weight edge in E that crosses the cut (V_1, V_2) and use this edge to unite the two MSTs on the subgraphs to make an MST of G. Either argue that the algorithm is correct or give a counter-example.
Solution: The algorithm is incorrect. I'm assuming the base case of the algorithm is when the size of the vertex set is 2, in which case the edge between the two vertices is returned. Here is a counter-example. Consider a square with edge weights 1, 2, 3, and 4. Suppose the first recursive call is on the subgraph containing edge 3 and the second recursive call is on the subgraph containing edge 4. Then those are the edges returned by the recursive calls. The non-recursive work adds edge 1 to give a spanning tree with edges 1, 3, and 4. But the MST consists of edges 1, 2, and 3.