**Carsten Hood** UIN: 922009787

CSCE 411-200 Fall 2016

Homework 4

Due Wed, Oct 26, 11:30 AM

—————————————

1. Exercise 17.4-3 (p. 471)

potential function:

First confirm potential function is valid:

, and since the result of an absolute value is always nonnegative we know that otherwise: .

**Case 1** – table is still at least full after a TABLE-DELETE operation:

**Case 2** – table becomes less than full after a TABLE-DELETE operation:

since we know that we can replace the absolute value signs:

then substitute the sizes (we know ):

So the amortized cost of deletion is and hence the total time for deletions is at most .

—————————————

2. Problem 17-5 (pp. 476-477).

(a) In the worst case each of the accesses seeks the last element in the sequence, so each of accesses traverses all elements, which takes per access andtotal.

(b) The cost of finding the element is and the cost of moving it to the front (through the previous nodes) is , hence ,

(c) As with move-to-front, finding an element takes its rank in the list, or . The cost of moving the element is equivalent to the number of transpositions, defined as .Hence the total cost is

(d) A transposition creates one inversion if the concerned pair of elements is already ordered, and undoes one inversion if they already constitute an inversion. Hence we have a change of inversion either way, and since this becomes

(e) ’s rank in either set can be stated as the number of elements preceding it plus one. The elements preceding either (1) precede in the other list as well or (2) follow in the other list. For these elements are defined in sets and , hence For these elements are defined in sets and , hence .

(f) Using move-to-front, we make new inversions by transposing before each of the prior elements. This also unmakes inversions as takes its proper earlier place relative to items in the other list. Using heuristic H, it is possible that one inversion is made for each of the transpositions. We add these changes to the potential together to obtain .

(g) We know that = 2;

and that .

We solve:

) // substitute

. // substitute

(h) We have shown that for the same lists. Consider a sequence of first-to-move operations , which can be rewritten; this sequence is less than or equal to a sequence of operations plus ; this is in turn less than or equal to 4 times the cost of the sequence of heuristic-based operations.

—————————————

3. Problem 21-3 (pp. 584-585).

a. Each node is visited once and set to BLACK just prior to iterating through its pairs. Since the second node in each pair to be visited will be set to WHITE when it is checked by the first node (since this second node will not have been visited and set to BLACK yet), the if-statement at line 9 will only (and always) be true the second time a pair is visited, i.e., exactly once.

b. The relevant operations here are *make-set*, which increments the set count by one, and *union*, which decrements the set count by one. One *make-set* operation is performed prior to calling *LCA* on a child of *u*, hence we have +1 set and then +1 depth. Before another child is considered in the first for-loop, we have a *union* operation, or –1 set and then –1 depth, correcting the set count for the next child or for unfolding up the recursion.

c. Nodes are visited from the bottom up as recursion unravels. The first time any given pair is considered, the first element is colored black but the second element will be white and hence the pair is skipped. The second (and final) time this second element is colored black, and since the first is already colored, the pair is processed. The lowest common ancestor is now available associated with either node, which are recently merged (via union) into one set, but only with subsequent nodes.

d. Let be the number of nodes. The first for-loop ensures *LCA* is called exactly once for each element. In *LCA*’s first two lines we have one *make-set* and one *find-set* for each element, so *make-set* operations and set operations. In the for-loop we have an additional *union* and *find-set* operation for each non-root element, or set operations. The second for-loop then compares each node to every other node in the tree and performs one *find-set* operation for each pair, or total. We have total set operations and exactly *make-set* operations. We substitute these values into the formula where is a slow-growing function defined in the afore-mentioned section, to obtain .

—————————————

4. Exercise 16.3-7 (p. 436).

“16.3-7 - 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.

Given set C of n chars, c occurs f[c] times

insert each c into a priority queue Q using f[c] as key

**while** |Q| > 1

x := extract-min(Q)

y := extract-min(Q)

z := extract-min(Q)

generate a new node v with left child x, middle child y, and right child z

f[v] = f[x] +f[y] + f[z]

insert (Q, v)

Proof:

We imitate the proof for a binary Huffman code by showing that the ternary algorithm also exhibits the greedy-choice and optimal-substructure properties. (Let C be set of characters c, each occurring f[c] times).

(1) To demonstrate the greedy-choice property:

Define three characters x, y, and z in C as having the lowest frequencies; first we show that these then optimally have identical lengths and differ only in their last bits. Consider 3 arbitrary sibling characters a, b, and c that are sibling leaves at maximum depth in T. Assume w.l.o.g. that f[a] <= f[b] <= f[c]. We can assume f[x] <= f[a], f[y] <= f[b], and f[z] <= f[c]. If f[x] = f[b] or f[x] = f[c], then f[a] = f[b] = f[c] = f[x] = f[y] = f[z], and this first lemma is clearly true. So assume f[x] /= f[b] and [x] /= f[c], and hence x /= b and x /= c.

If we swap a with x, b with y, and c with z, such that x, y & z hold the lowest positions, then the cost cannot increase, because x, y, and z have minimum frequencies; thus the new tree so-formed is also optimal. This exhibits the greedy-choice property.

(2) To demonstrate the optimal-substructure property:

Using the earlier definition for C, x, y, and z, let C’ be equal to C with x, y, and z replaced by a new character v such that f[v] = f[x] + f[y] + f[z]. If T’ is a tree representing the optimal prefix code for C’, the let T be the tree obtained by replacing v with a node having x, y, and z as children. We will show that T represents an optimal prefix code for C.

All characters c common to T and T’ have identical costs, i.e., d(c) = d’(c). Since d(x) = d(y) = d(z) = d’(v) + 1 we can say that:

f[x]d(x) + f[y]d(y) + f[z]d(z) = (f[x] + f[y] + f[z])(d’(v) + 1) = f[v]d’[v] + f[x] + f[y] + f[z]

which means that: d(T) = d’(T) + f[x] + f[y] + f[z].

By contradiction: if T does not represent an optimal prefix code for C, then an optimal tree T” exists such that d”(T”) < d(T). If siblings x, y, z in T” are replaced by a node v with x, y, and z as children (as described above) then the resulting tree must have a cost less than d(T’), which is a contradiction.

—————————————

5. Exercise 23.2-8 (pp. 637-638).

The algorithm fails; as a counter-example, consider the below graph:

A 1 B

| 3 | 2

C 10 D

If the algorithm partitions the vertices into sets V1 = {A, B} and V2 = {C, D} then the respective minimum edges are clearly the only edges in these subtrees, i.e., (A,B) and (C,D), and then the minimum edge that unites the two trees is (B,D). The resulting spanning tree is the tree A 1 B 2 D 10 C with a total weight of 13, which is clearly incorrect; the correct minimum-spanning tree swaps the edge (C,D) of weight 10 with that of (A,C) of weight 3.