Assignment 3

Sanjay G. Rao

March 20, 1998

Problem 1:
We define two charges: the path charge and the root charge. We will show that the path charge is $O(n*logn)$ and the root charge is $O(m)$, thus giving the $O(n*logn+m)$ bound. For any given L-Find operation, we assess a path charge of 1 unit per node traversed (the node to be found, and each of its anscestors), except the child of the root. For each L-Find we assess 1 unit of root charge corresponding to the child of the root. Clearly, m L-Find operations result in a charge of m to the root charge. To bound the path charge, consider the maximum path charge that any particular node can ever be assessed. Observe that each time the node is assessed a path charge of 1, the distance of the node from the root decreases by 1/2. (Note that when the node is a child of the root, no path charge is assessed. Also, once a node is a child of the root, it always remains child of the root, and no more path charge is ever assessed.) The distance of the node from the root could be atmost n to begin with. After k assessments of path charges, the cost is $n/2^k$. But, $n/2^k \geq 1$ and hence, $k \leq logn$. Thus any given node can atmost be charged $logn$ path charges. The total path charge across all nodes is then $n * logn$.

Problem 2:
Notation:
message(i,j): The substring of the message beginning at character i and of length j.
Prev[1..m+1]:Array of $m + 1$ elements.

```
1. Initialize each element of Prev[2..m] to UNDECIDED.
2. for (i=1;i<=m;i++)
        3. If (Prev[i]==NOT_A_STARTING_WORD || Prev[i]=UNDECIDED) continue;
           (back to loop at Step 2)
        4. for(j=1;j<=l;j++)
                5. if message(i,j) is a codeword then
                      6. if(Prev[i+j] != UNDECIDED)
                                 Prev[i+j]=NOT_A_STARTING_WORD
                              else
                                 Prev[i+j]=i
        7. end Step 4 for loop.
8. end Step 2 for loop.
9. for(i=m+1; i > 1; i=Prev[i])
        print Decode(message(Prev[i],i-Prev[i])).
```

The algorithm prints out the decoded message in reverse order. It works as follows. $Prev[i]$ contains the position where the previous word of the decoded message starts, assuming that the next word starts at position i. At the end of Step 8, $Prev$ contains the starting position of all words and decoding the message in Step 9 is a simple matter of following the links backwards beginning from $m + 1$.

The array $Prev$ is constructed in Steps 3-8. For each starting position i, Steps 4-6 determine possible starting positions of the next word. If the substring of the coded message beginning at i and of length j $(1 \leq j \leq l)$ is a code word, then, $i+j$ is a possible starting position of the next word. Step 6 needs some explanation. If $Prev[i+j] = k$ $(\neq UNDECIDED)$ then it implies that it is possible to parse the message from character 1 to character $i+j-1$ in two different ways (the last word in the different ways starting at k and i). If it were then possible to parse the message from position $i+j$ to position m, then, we have two distinct ways of parsing the entire message, but this is not possible as the code is uniquely decipherable. Hence, a word cannot start at position $i+j$.

The complexity of the algorithm above is $O(ml^2)$. The loop in Step 2 executes m times, the loop in Step 4 executes l times, and Step 5 (checking if a given substring is a codeword) takes l time. Preprocessing required is to organize the codewords as a binary tree T. It is possible to improve the code in the loop at Step 4, so that it runs in $O(l)$ time and the entire algorithm takes $O(ml)$ time. This can be achieved by traversing the substring of l characters (beginning at i) in the tree T and marking all valid codewords along the path.

Problem 3:
We show an upper and lower bound of $O(logn)$. To show the upper bound, observe that to increment the value of a node, delete the node, and reinsert it with the higher value. Delete takes $O(logn)$ and insert takes $O(1)$ and we are through. To show the lower bound, we show that if we can do increment better than $O(logn)$, then, we can perform sorting in amortized time better than $O(n * logn)$. Sorting can be done by doing:
findmin;increment this min value by infinity;
and repeating the above operations n times. Findmin takes $O(1)$ time, and if increment takes $O(f(n))$ time, then, sorting can be done in $O(n * (1 + f(n))$ time, that is $O(n * f(n))$ time. We can't sort better than $O(n * logn)$ and hence, $f(n)$ cannot be better than $logn$.