CS 180 - Homework 3

Darren Tsang, Discussion 1I

Produced on Saturday, May. 23 2020 @ 03:01:10 PM

Question 1

remove the edge $e = (u, v)$ with weight changed from w to w' to obtain T_1 and T_2
let $C = \{\text{weight of edges} \in G \text{ with one end in } T_1 \text{ and one end in } T_2\}$
return $(w' < c,\ \forall c \in C)$

Edge e was originally in T, which meant that out of all edges with one end in T_1 and one end in T_2, it had the lowest weight. After the weight of e was changed from w to w', we need to check if w' is still the lowest weight out of all edges with one end in T_1 and one end in T_2. If it is still the lowest, then T is still a minimum spanning tree, else it is no longer a minimum spanning tree.

Additionally, we are only looking at the weight of the edges. At worst, we would have to look at the weight of every edge, which means it is $O(m)$.

Question 2

Algorithm A: $T(n) = 9T(\frac{n}{3}) + n^2$, $\frac{9}{3^2} = 1 \rightarrow T(n) = O(n^2 log(n))$

Algorithm B: $T(n) = 2T(n-1) + n^0$

In this case, $T(n)$ is not in the correct form for Master's Theorem. Thus, we will look at the recursion tree produced. $T(n)$ splits into two $T(n-1)$, and each splits into two $T(n-2)$ (for a total of 4 $T(n-2)$), etc. This process continues until there are n levels, thus $T(n) = O(2^n)$.

Algorithm C: $T(n) = 5T(\frac{n}{2}) + n^1$, $\frac{5}{2^1} = 2.5 > 1 \rightarrow T(n) = O(n^{log_2(5)})$

Question 3

findLocalMin(node n){
  if(n has children){
    $left$ = left child of n
    $right$ = right child of n

    probe $left$ to determine x_{left}
    probe $right$ to determine x_{right}
    probe n to determine x_n

    min = minimum(x_{left}, x_{right}, x_n)

    if($min = x_n$){
      return n
    } else if ($min = x_{right}$){
      return findLocalMin($right$)
    } else if ($min = x_{left}$){
      return findLocalMin($left$)
    }
  } else{
    return n
  }
}

The algorithm above essentially tries to see if the labels of the children of a given node n is less than the label x_n. If there are no children with a label less than x_n, then we know that x_n is a minimum. We do not have to check if the parent v of n has a label less than x_n because of the way the algorithm is designed; we already know that $x_n < x_v$, else findLocalMin(n) would have never been called. The case for the root and leaves are trivial, as the root has no parent (so there's no way the parent has a lower value than the root) and the leaves have no children (so we can just return the leave itself).

The algorithm uses only $O(logn)$ probes to nodes of the complete binary tree T because for every function call, the tree essentially splits into two; it will either continue on in the left subtree or the right subtree, but never both. This idea is similar to that of binary search on a sorted array; every function call results in the problem getting cut down by half.

Question 4

overlap($[s_1, f_1], [s_2, f_2]$){
  if($f_1 < s_2$){ return 0 }
  else if ($f_1 \leq f_2$){ return $f_1 - s_2 + 1$ }
  else if ($f_1 > f_2$){ return $f_2 - s_2 + 1$ }
}

sort intervals by s_i in increasing order to obtain the ordering $L = \{[s_1, f_1], [s_2, f_2], ..., [s_n, f_n]\}$
largestOverlap(List L){
  n = number of intervals in L

  middle = floor($\frac{n}{2}$)
  firstHalf = $\{[s_1, f_1], ..., [s_{middle}, f_{middle}]\}$
  secondHalf = $\{[s_{middle+1}, f_{middle+1}], ..., [s_n, f_n]\}$

  if (n = 0 or 1) {
    return 0
  } else if (n = 2) {
    return(overlap($[s_1, f_1], [s_2, f_2]$))
  } else{
    firstHalfMax = interval where f_i is max in firstHalf
    cross = max({overlap(firstHalfMax, interval) | interval $\in$ secondHalf})

    return(max(largestOverlap(firstHalf), largestOverlap(secondHalf), cross))
  }
}

overlap() is a simple helper function that calculates the overlap between two intervals with the assumption that $s_1 \leq s_2$.

The algorithm above begins with what the hint suggested; sort the intervals in increasing order based on s_i values, then splits the set of intervals into the firstHalf and secondHalf. Then, largestOverlap(firstHalf) and largestOverlap(secondHalf) is called recurseively until the set firstHalf and secondHalf only have two intervals left in each. Once there are only two intervals left, overlap() is called to calculate the overlap between the two intervals. If there are 0 or 1 intervals in the set passed to largestOverlap(), 0 will be returned because you must have at least two intervals for there to be an overlap. Additionally, there is a variable *cross* that deals with "linking" the subproblems together; it accounts for the fact that the largest overlap may occur between an interval in firstHalf and an interval in secondHalf.

In the algorithm, sorting is $O(nlogn)$; we only need to sort once because once it is sorted, it will never become unsorted. Also, the reccurence can be written as $T(n) = 2T(\frac{n}{2}) + n \rightarrow T(n) = O(nlogn)$. Thus, the overall algorithm is $O(nlogn)$.