Homework 3

Raymond Lin, 304937942

May 23, 2020

Problem 1

Idea:
We have an MST of a graph G, and we are asked to find whether the MST will still be an MST, if we exchange the weight, w, or an edge (u, v), with w'. We can solve this problem by splitting the MST into two components and observing the edges that connect these two components. More specifically, let us remove the edge (u, v) from the MST. By definition, an MST is connected and does not have any cycles, or in other words, there is only one path from a particular node to any other node. Thus, if we remove (u, v), the MST will be split into two components. Let us call these components $C1$ and $C2$, where $u \in C1$ and $v \in C2$. Then, if we run BFS, on both of these nodes, we can identify which component each node belongs to. Now, we can loop through all the edges of the G, and compare only the edges that connect $C1$ and $C2$. This is because only these edges will create a spanning tree by connecting itself with $C1$ and $C2$. The edge with the lowest cost is the one that belongs to the new MST. If the lowest edge is still (u, v), then the new MST is the same as the old one. If not, then it is not.

This algorithm runs in $O(m)$ time, because we traverse all the edges a total of 3 times. The first two traversals come from running BFS twice to generate the array that holds each node's component. The last traversal comes from search through the edges in G for all the edges that connect the two components.

Algorithm:

```
let weight(x,y) = the weight of edge (x,y)
let M = MST of G with edge (u,v)
Remove (u,v) from M
let C1 = the component of M such that u ∈ C1
let C2 = the component of M such that v ∈ C2
let label[i] = 1 ∀ nodes i ∈ C1                          // can do this with BFS
let label[j] = 2 ∀ nodes j ∈ C2                          // can do this with BFS

for each edge (s,t) in E:
  if label[s] ≠ label[t] and weight(s,t) < weight(u,v):
     return false
return true
```

Problem 2

a)

$T(n) = 9 \times T(\frac{n}{3}) + cn^2$

$a = 9, b = 3, k = 2$

$r = \frac{a}{b^k} = \frac{9}{9} = 1$

$\rightarrow O(n^2 log_3 n)$

b)

$T(n) = a \times T(\frac{n}{b}) + cn^k$

size of subproblem $= n - 1$

$n - 1 = \frac{n}{b}$

$b = \frac{n}{n-1}$

$a = 2, b = \frac{n}{n-1}, k = 0$

$\rightarrow$ can't use master theorem because b is not constant

$T(n) = 2 \times T(n-1) + c$

$T(n-1) = 2 \times T(n-2) + c$

$T(n) = 2 \times (2 \times T(n-2) + c) + c$

$= 2^2 \times T(n-2) + 3c$

$\cdots$

$= 2^n \times T(0) + (2^n - 1) \times c$

$= 2^n + (2^n - 1) \times c$

$= O(2^n)$

$\rightarrow O(2^n)$

c)

$T(n) = a \times T(\frac{n}{b}) + cn^k$

$a = 5, b = 2, k = 1$

$r = \frac{a}{b^k} = \frac{5}{2^1} = \frac{5}{2} > 1$

$\rightarrow O(n^{log_2 5})$

Problem 3

Idea:
The idea for this algorithm is that each path from the root node to any leaf node must have at least 1 local minimum, if the graph was just that path itself. This fact is true for the following reasons. Because the values of each node are distinct, there must be some increasing or decreasing relationship between any pair of adjacent nodes in the path. For 1 local minimum in the path, there are three cases. The first case is when the values in the path are strictly decreasing. Then, the only local minimum is at the leaf. The second case is when the values are strictly increasing. Then, the only local minimum is at the root. The third case is when the values decrease and then increase. The local minimum is at the point where the values start to increase. Any other configuration of increasing/decreasing values will result in more than 1 local minimum.

We have not considered the case where the found node from above has another child that is smaller itself. For example, if the found node's parent and left child were greater than itself, but its right child was not, then this disqualifies it from being a local minimum. We can fix this by traversing the child whose value is less than the current node's value. This means that at every step in our algorithm, we know that the current node's parent is always greater than the current node. At each iteration, we must only check the node's left and right children. If both children are less than the current node, it is arbitrary which one we traverse. If both children are greater than the current node, then we have already found our local minimu, since the parent will always be greater than the current node. We stop when we have found a local minimum or we reach a leaf. The leaf must be a local minimum, because we know its parent is greater than itself, and by definition of a leaf, it has no children.

This algorithm runs in $O(logn)$ time, because in the worst case, we are traversing a single path from the root of the tree to one of the leaves. In other cases, it will have found the local minimum before reaching the leaf.

Algorithm:

```
let root = root of our tree

// Recursive helper function
def findLocalMin(node):
   if node.left = null and node.right = null:
      return node

   if node.left.value > node.value and node.right.value > node.value:
      return node

   if node.left.value < node.value:
      return findLocalMin(node.left)
   else:
      return findLocalMin(node.right)


// Call the function
findLocalMin(root)
```

Problem 4

Idea: The idea of this algorithm is to partition the intervals into two half sets, and recursively call a function to find the largest overlap in each of these two half sets. The missing case is when there is an interval from one set that overlaps with an interval in the other set, and this overlap has a larger value than the two values returned from th recursive calls. We take the maximum of these three values to be the maximum overlap between any two intervals.

This algorithm runs in $O(nlogn)$ time, because we must first sort the intervals by their start times. Once we do this, the other time is spent on finding the largest possible overlap between intervals in different half sets, at each recursive call. We can do this by first finding the interval with the latest finish time in the first half set, L_i. We can do this in linear time. Then, we can search the second half set for intervals whose start time is before that of L_i. These intervals are guaranteed to overlap with L_i, so we can find the interval that has the largest overlap with L_i. We can do this in linear time using linear search through the second half set. The time complexity of this part is also $O(nlogn)$, because at each level of recursion, we check n elements, and there are $logn$ levels of recursion. Thus, the total time complexity is still $O(nlogn)$.

Algorithm:

```
let S = our set of intervals
S.sort() // by starting time

// Recursive helper function
def findMaxOverlap(X: a sorted set of intervals):
   if X.size() = 0 or X.size() = 1:
      return 0

   let L = first half of X
   let R = second half of X

   // find the interval with the latest finish time in first half set, L
   let Li = the interval in L with the latest finish time, initialized to L[0]
   for each interval, u, of L:
      if u.endTime > Li.endTime:
         Li = u

   // find the interval in R with the largest overlap with Li
   let centerMaxOverlap = 0
   for each interval, v, of R:
      if v.startTime > Li.endTime
         break
      // find the overlapping parts of intervals Li and v
      let overlappingInterval = min(v.endTime, Li.endTime) - v.startTime
      if overlappingInterval > centerMaxOverlap:
         centerMaxOverlap = overlappingInterval

   // find the largest possible overlap in left half set
   let leftMaxOverlap = findMaxOverlap(L)
   // find the largest possible overlap in right half set
   let rightMaxOverlap = findMaxOverlap(R)

   return max(leftMaxOverlap, rightMaxOverlap, centerMaxOverlap)

// call the function
findMaxOverlap(S)
```