CS210-Theoretical Assignment 2

Shubham Agarwal 13674
Vikas Jain 13788
Dept of Computer Science and Engineering
IIT Kanpur

October 20, 2014

Solution 1 :

To design an O($\log n$)time algorithm, The tree would be splitted in O($\log n$) trees and then these trees will be merged using modified special_union function .

Modified Special Union

This function will take four inputs.Pointers to trees T1 and T2, diffrence D of their black heights and the value x such that x lies between T1 and T2.

In this function, modification is that we'll not delete the node from the tree. Instead node through which two trees have to be merged is already provided as input to function.Also, we'll not compute black height of two trees.Their diffrence is also taken as an input.To find the location of merging,one has to traverse black nodes same as diffrence. This will help reducing time complxity of Algorithm.

Pseudo Code :

Mod_Union(T_1, T_2, D, x)
{

Assume Black Height of T_1 >= Black Height of T_2
Keep following pointer of T_1 until we reach a node v such that :
1. Left(v) is black.
2.Subtree T rooted at v has same Black Height as that of T(by traversing upto the diffrence of black heights).
Left(x) $\leftarrow T_1$
Right(x) $\leftarrow T$
Color(x) $\leftarrow$ red
Left(v) $\leftarrow$ x
parent(x) $\leftarrow$ v
If color(v) is red, remove color imbalance

}

Split

Find Predecessor of x in T. Then start traversing upward from there upto root. If you traverse rightward from a node v, then left sibling of v will be splitted . This will be then merged with tree of similar type(i.e. $T <= x$). While merging one has to know the black height.Required Black Heights will be stored in few variables.

Pseudo Code :

```
Split(T,x)
{
        y ← x;
        x ← predecessor(x);
        Compute Black height of subtree rooted at x;
        bh_x ← Black height of subtree rooted at x;
        Traverse tree from root to a node, say p, storing value x;
        T1 ← left(p);
        T2 ← right(p);
        Calculate Black height of T1 and T2 with help of x
        ( if(x is red) bh_T1 = bh_T2 = bh_x;
        else bh_T1 = bh_T2 = bh_x − 1; )

        if(T1 is red ) make T1 black;
        bh_ T1 ++
        if(T2 is red ) make T2 black;
        bh_ T2 ++
        while(parent(p) <>NULL)
        {
                if(p is right child)
                {
                        temp = left sibling of p;
                        bh_temp = bh_p;
                        if(temp is red ) make temp black;
                        bh_temp ++;
                        T1=Mod_Union(temp,T1,bh_temp - bh_ T1,value(parent(p)
                        );
                }
                else
                {
```

temp= right sibling of p
bh_temp=bh_p
if(temp is red) make temp black;
bh_temp ++;
T_2=Mod_Union(temp, T_2, $bh\_temp - bh\_T_2$,
value(parent(p));

}
if(p is black) bh_p ++;
p ← parent(p);
}
if(x!=y) Insert(x,T_1);
return T_1 and T_2;
}

Time Complexity of Mod_Union :

Traversing upto same black height = **O**(diffrence of black height of two trees)
Removing color imbalance = **O**(diffrence)
Rest will take **O**(1) time
Overall order = O(diffrence of black heights of two trees)

Time Complexity of Split :

Finding predecessor $\longrightarrow$ $\mathbf{O}(\log n)$
Computing bh_x $\longrightarrow$ $\mathbf{O}(\log n)$

Time Complexity of overall while loop :

Suppose i_{th} iteration takes T_i time.There are k iterations Then :
T=$T_1 + T_2 + T_3 + ... + T_k$
Notice that in each loop program operates either under if condition or under else condition. Let A be the set of iterations when program is operating under if conditions. Similarily B be the set of iterations when program is operating under else condition.Then :
A=$\{ T_{a_1}, T_{a_2}, T_{a_3} ... T_{a_p}\}$
B=$\{ T_{b_1}, T_{b_2}, T_{b_3} ... T_{b_{k-p}}\}$
$\sum_{i=1}^{p} T_{a_i} + \sum_{i=1}^{k-p} T_{b_i} = \sum_{i=1}^{k} T_i = \mathrm{T}$
$\sum_{i=1}^{p} T_{a_i} = \sum_{i=1}^{p-1} \mathbf{O}$(diffrence of black height of T_{a_i} and $T_{a_{i-1}}$) = $\mathbf{O}(\log n)$
$\sum_{i=1}^{p} T_{b_i} = \sum_{i=1}^{k-p-1} \mathbf{O}$(diffrence of black height of T_{b_i} and $T_{b_{i-1}}$) = $\mathbf{O}(\log n)$
$\Longrightarrow$ T = $\mathbf{O}(\log n)$

Rest are of **O**(1)
So overall time complexity of Split = $\mathbf{O}(\log n)$

Justification:

In **Special Union Algorithm**, we have removed the deletion operation. The purpose of deletion of node was to obtain a node x such that x lies between the values of two trees. Instead of obtaining it from a tree, we have provided it as an input to the function. So, it should work correctly.
In **Split** function, starting from node about which we have to split we keep traversing upward. At each upward step, we keep splitting the tree into smaller and bigger part. As soon as, we get two smaller/bigger trees, we immediately merge them using special union function. Since Special Union is working correctly, all we need to check that input provided to special union function is correct.
T_1, T_2 : We should check that T_1 and T_2 are proper red black tree. Since both T_1 and T_2 are subtrees, we just need to make the color of their root to black. Program does this each time when it splits a tree. Each right subtree in the path is merged with T_2(smaller one) and left wth T_2. So T_1 and T_2 are proper red black trees, either both smaller one or both larger one.
Diffrence and x : It can be clearly seen from code that both diffrence and value of x are properly calculated.
Hence, inputs given to Mod_Union are correct. So, algorithm work properly.

Solution 2

1.To find $min\_L(s)$ for a vertex $s \in V$, single DFS Traversal from s can be revoked in the algorithm. For each vertex encountered in DFS Traversal from s, store in $min\_L(s)$, the minimum of $L(v)$ for all v in the DFS Traversal. Also there is no change in DFS Traversal for directed or undirected graph since adjacency list will be of precise form of neighbors only.

psuedo code:

```
DFS(v){
        visited(v)←true;
        if(min_L(s) > L(v)) min_L(s) = L(v); //Global min_L(s)
        For each neighbor w of v{
                if(visited(w)=false)
                DFS(w);
        }
}
```

```
Find_minLabel(G){
        For each v ∈ V
        visited(v)=false;
        min_L(s) = L(v);
        //Global min_L(s) DFS(s);
        return min_L(s);
}
```

Time Comlexity:

The above algorithm will invoke only one DFS from the node(say s) of which $min\_L(s)$ is required. Hence the complexity is $O(m + n)$.

Justification:

Since DFS(s) will call DFS(w) for each w reachable from s(DFS Traversal) hence at last $min\_L(s)$ will get the required value.

2.To solve the problem, a new adjacency list A is maintained such that for each vertex $v \in V$, the neibhor list of vertex v contains all those nodes which have edges falling on v(In simple words reverse the direction of all edges in G and obtain corresponding adjacency list). Now in a separate array we can sort vertices according to their label value.From in this sorted order DFS Traversal is called according to the new adjacency list and $min\_L(v)$ is computed accordingly.
psuedo code:

Considering vertices in V are numbered from 0 to n-1.

```
Find_min(G){
        Create new Adjacency list A of size n;
        For each node v in graph G{
        for each neighbor w of v{
                Add v to neighbor of w in A;
        }
        }
        Create an array Label[n][2] which store vertex number and corresponding Label.
        //Label[i][0]=vertex no., Label[i][1]=L(vertex)

        Sort Label[ ][2] in ascending order of Label[i][1];
        Create array min_L[n];//Output, Globally accessible
        For(i=0 to n-1){
        s ← Label[i][0];
        if(visited(s)=false) {
        min_Label = Label[i][1]; //Updating global varaible min_Label;
        DFS(s);
        }
}
```

```
DFS(v){
visited(v)←true;
min_L(v) ← min_Label;
For each neibhor w of v in A{
        if(visited(w)=false)
        DFS(w);
}
}
```

Complexity:

Creating new Adjacency list will take $O(m+n)$ time. Now Sorting the array Label of size n will take time $O(nlogn)$. Now DFS() in adjacency list A will take $O(m+n)$ time. Hence the total time complexity is $O(m+nlogn)$.

Justification:

All the vertices in new adjacency list will get visited during DFS.
Claim: During DFS(v), $min\_L(v)$ is computed correctly.

Consider the starting vertex(say u) in Find_min() for which DFS(v) is called. Now since in Find_min DFS() is called in ascending order of label. It implies v is not reachable from any of vertex before u in sorted order otherwise v must have visited. Hence label of u is smallest possible label v can reach to in original adjacency list. And now in new adjacency list A, u can reach to v and $min\_L(v)$ stores the label of u.

Solution 3

The stated problem can be solved using TWO DFS Traversal. Firstly invoke DFS Traversal for any vertex $v \in V$(Let Tree be a graph G(V,E)).The DFS Traversal is modified in such a way that it will compute the vertex farthest from the starting vertex v of DFS Traversal. Now the vertex obtained from the first Traversal must be one of the extreme vertex of one of the longest length path. Now apply DFS Traversal from the vertex obtained and it will compute the farthest vetex from it and the path will be longest.

Pseudo Code:

```
DFS(v){
visited(v)←true;
t++; //t,node and len are Global Variables.
if(t > len){
len = t;
node = v;
}
For each neighbor w of v{
        if(visited(w)=false)
        DFS(w);
}
t − −;
}
```

```
Find_LongestPath(G){
        t ← −1;
        len ← −1;
        For any v ∈ V
        DFS(v);
        t ← −1;
        len ← −1;
        DFS(node);
return len;
}
```

Complexity:

There are n nodes in the tree and since it is a tree the number of edges is $n-1$. In the above algorithm the time taken will be time taken by 2 DFS Traversal. Hence complexity is $O(m+n)$. Now for a tree $m = n-1$, hence time complexity is $O(n)$.

Justification:
Justification of 2 points will do.

1. DFS will give the node which is farthest.

Since DFS(v) will reach all the nodes reachable from it, focus on the time when DFS(v) reaches its farthest node. Now since for each step in depth, t is increamented in each step and for farthest node it is maximal. Hence *len* and *node* is updated for maximal t. This verifies that *node* will be the farthest node from v and *len* is the length of path between v and *node*.

2. After first DFS(v) for any v, *node* is the extremum vertex of the one of the longest path possible in tree.

This can be proved using Contradiction. Suppose First DFS(v) doesn't land to the extremum end of any longest path. Then there exist *node* vertex and from v, it is maximum length path. Now consider the path of maximum length in the tree and the extreme vertices of this path are u and p.The path from v to *node* must be connected to the path from u to p.
Consider the figure 1

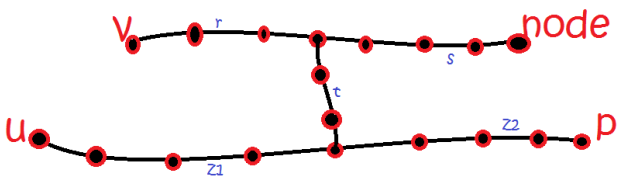

Figure 1: First DFS Traversal

$r, s, t, z1, z2$ are the correponding path lenght shown.
Consider $z1 > z2$ (without loss of generality), now since v to *node* is maximal length path from v.
$\Rightarrow r + s \geq r + t + z1$
$\Rightarrow s \geq t + z1$ and $t \geq 0$.
Hence u to *node* is a path of lenght greater than path length from u to p. Hence it is a contradiction and *node* must be a extremum vertex of longest possible path.