CS345A Algorithms-II

Assignment-2

Deepanshu Bansal (150219)
Mukul Chaturvedi (150430)

Q1 Orthogonal Range Counting

Here we store the points as discussed in class. That is create a BST from x coordinates and augment each node with tree according to y coordinate.

For this problem let's consider for 1-Dimension ie. if we are given only x coordinates we can find the number of points in the range x_i and x_j by simple binary search.

Here at each node of tree we are storing **"size"** field which gives us the size of the subtree rooted at that node so that we can easily find the rank of a node or point in tree. Thus if we want to find the number of nodes between two coordinates just find the largest smallest elements from x_i and x_j and their ranks and difference between ranks (+1 , -1 adjusted) gives us the number of points in range x_i and x_j .

Our function **isPresent**(w, x_1 , x_2 , y_1 , y_2) returns 1 if w is present in that rectangle otherwise zero.

T(**isPresent(...)**) = **O(1)**

Our function **count**(right(u) , y_1 , y_2) returns the number of points in the range y_1 , y_2

T(**count(...)**) = **O(logn)**

And now for 2-D just first find **LCA(x_i, x_j)** and now there will be O(logn) nodes between x_i and LCA and also between x_j and LCA. It is ensured that all of these nodes have all points whose coordinates are between x_i and x_j . And for each of these O(logn) nodes we apply the above method of counting in their respective augmented y-trees to find the points in the range y_i and y_j and add result of each node to the final result thus this process ensures that we covers all the points in range of rectangle (x_1 , x_2 , y_1 , y_2) in time **$O(\log^2 n)$.** Now for the size of data structure is **O(nlogn)** as discussed in class.

RangeCount(T, x_1 , x_2 , y_1 , y_2){

Let u be the node storing x_1 element;

Let v be the node storing x_2 element;

finalResult ← 0 ;

w ← **LCA**(u, v); finalResult ← finalResult + **isPresent**(w, x_1 , x_2 , y_1 , y_2);

if(u<> w){

finalResult ← finalResult + **isPresent**(u, x_1 , x_2 , y_1 , y_2);

if (right(u)<>NULL) finalResult ← finalResult + **count**(right(u) , y_1 , y_2);

While(parent(u)<> w){

if(u=left(parent(u))){

finalResult ← finalResult + **isPresent**(parent(u), x_1 , x_2 , y_1 , y_2);

if(right(parent(u)<> NULL) finalResult ← finalResult + **count**(right(parent(u)) , y_1 , y_2);

}

u ← parent(u);

}}

if(v<> w){

finalResult ← finalResult + **isPresent**(v, x_1 , x_2 , y_1 , y_2);

if (left(v)<>NULL) finalResult ← finalResult + **count**(left(v) , y_1 , y_2);

While(parent(v)<> w){

if(v=right(parent(v))){

finalResult ← finalResult + **isPresent**(parent(v), x_1 , x_2 , y_1 , y_2);

if(left(parent(v)<> NULL) finalResult ← finalResult + **count**(left(parent(v)) , y_1 , y_2);

}

v ← parent(v);

}}

return finalResult;

}

Q2 Flip bits

Create a BST from the sequence as discussed in class such that inorder traversal of BST makes our sequence. Augment BST by maintaining a **"size"** field along each node which keeps track of size of subtree rooted at that node. For performing Multi-flip(i, j) we need one more additional field **"flipped"** for each node such that if its value is true that would mean value of all the nodes in its subtree have to be flipped. **flip** is a function which just invert the bit given to it ie. 1 → 0 and 0 → 1.

All our operations takes **O(logn)** time.

Insert, Report operation → We traverse only one time at max through the tree and hence only steps proportional to the height of tree ie O(logn).

For **Delete** operation we are taking constant time in some cases or in worst case finding the predecessor which again takes only steps proportional to height of tree. Thus O(logn) time complexity.

For **Multi-flip** operation we are first finding **LCA**(lowest common ancestor) which takes O(logn) time and then after that for each i and j we can at most traverse to the root taking steps proportional to height of tree which again takes O(logn) time. Thus total time will be **O(logn)**.

Our function **flip(b)** take **O(1)** time.

```
Insert(i, b){
        return Insert(i, b, D, false);
}
Insert(i, b, D, toBeFlipped) {
if(D == NULL) {
        create a new node u;
        val(u) ← b; size(u) ← 1; flipped ← false;
        if(toBeFlipped==true){
                val(u) ← flip(b);
        } else {
                valu(u) ← b;
        }
        left(u) ← NULL ; right(u) ← NULL ;
        return u;
} else{
        size(D) ← size(D) + 1;
        if(flipped(D)==true){
                toBeFlipped = !toBeFlipped ;
        }
        if(left(D)=NULL) s ← 0;
        else s ← size(left(D));
        if( i ≤ s +1) left(D) ← Insert(i, b, (left(D)), toBeFlipped);
        Else right(D) ← Insert(i-s-1, b, (right(D)), toBeFlipped);
        return D;
    }
}
```

Report(i){

found=false; u ← T ; toBeFlipped ← flipped(u);

While(not found){

if(left(u)=NULL) s ← 0;

Else s ← size(left(u));

if(s = i − 1) found ← true;

else if(s > i − 1) u ← left(u) ;

else {

i ← i − size(left(u)) − 1 ;

u ← right(u);

}

if(flipped(u)==true){

toBeFlipped = !toBeFlipped ;

}

}

if(toBeFlipped==true){

return **flip**(val(u));

} else {

return val(u);

}

}

Delete(i){

Let u be the node storing ith element;

if(u is leaf node){

    Just delete it and return;

}

Else if(u has only one child){

    Just remove it from the tree and join its parent to its child

    if(left(parent(u) == u)) {

        if(flipped(u)==true) flipped(child(u)) ← !flipped(child(u));

        left(parent(u)) ← child(u);

    }

    //similarly if u is right child of parent

}

Else if(u is an internal node){

    // let v be its predecessor that we will find

    toBeFlipped ← false;

    v ← left(u)

    if(flipped(left(u)) == true) toBeFlipped ← !toBeFlipped;

    while(right(v) != NULL){

        if(flipped(right(v)) == true) toBeFlipped ← !toBeFlipped;

        v ← right(v);

    }

    if(toBeFlipped==true) val(v) ← **flip**(val(v));

    val(u) ← val(v);

    free(v);

}

}

Multi-flip(i, j){

Let u be the node storing ith element;

Let v be the node storing jth element;

w ← **LCA**(u, v); val(w) ← **flip**(val(w))

if(u<> w){

val(u) ← **flip**(val(u));

if (right(u)<>NULL) flipped(right(u)) ← !flipped(right(u));

While(parent(u)<> w){

if(u=left(parent(u))){

val(parent(u)) ← **flip**(val(parent(u)));

if(right(parent(u)<> NULL) flipped(right(parent(u))) ← !flipped(right(parent(u)));

}

u ← parent(u);

}

}

if(v<> w){

val(v) ← **flip**(val(v));

if (left(v)<>NULL) flipped(left(v)) ← !flipped(left(v)) ;

While(parent(v)<> w){

if(v=right(parent(u))){

val(parent(v)) ← **flip**(val(parent(v)));

if(left(parent(v)<> NULL) flipped(left(parent(v))) ← !flipped(left(parent(v)));

}

v ← parent(v);

}

}

}

Q3 Synchronizing the circuit

(a) The optimal solution for a node(T) is optimal solution of left child plus optimal solution of right child plus absolute difference between max_delay in left subtree(T_1) and max_delay in right subtree(T_2).Thus,if we have optimal solution of left tree T_1 and optimal solution of right tree T_2 we can have optimal solution of T. So greedy step will be following

$$\textbf{OpSolution(T) = OpSolution}(T_1) + \textbf{OpSolution}(T_2) + |\textbf{maxDelay}(T_1) - \textbf{maxDelay}(T_2)|$$

This step transforms the problem of T (given instance) into two smaller instances T_1 and T_2 .

(b) Again relation between T and smaller instances is

$$\textbf{OpSolution(T) = OpSolution}(T_1) + \textbf{OpSolution}(T_2) + |\textbf{maxDelay}(T_1) - \textbf{maxDelay}(T_2)|$$

The global variable ans stores the minimum delay to synchronize the circuit. Now, we know that **every node of circuit is synchronous** in the final solution so if node is a leaf then there is no need to synchronize. For, other nodes in circuit we first find maximum delay in left subtree and in right tree. We add difference of the maximum delay in the two subtrees to ans and then recursively call sync function on left child and right child and also add these values to ans. In one call sync function call takes O(1) time without the recursive calls. All the nodes of circuit is called only once by sync function so the total time of algorithm is O(n) where n is the nodes in the circuit.

Proof of Correctness

$D_L(u)$: max delay along any leftward path

$D_R(u)$: max delay along any rightward path

If ($D_L(u) \geq D_R(u)$) increase delay of right edge by $D_L(u) - D_R(u)$

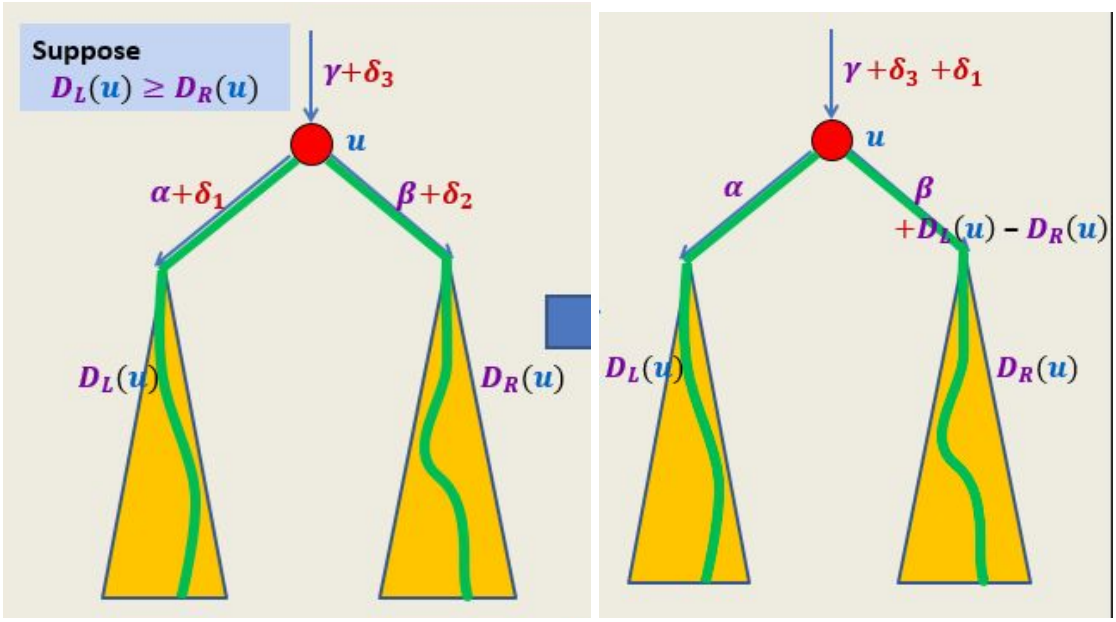
Else increase delay of left edge by $D_R(u) - D_L(u)$

Now we have to prove that solution is optimal if delay enhancement by $u = |D_R(u) - D_L(u)|$.

In the optimal solution, the delay along any path from u to any leaf node is $\max(D_R(u) , D_L(u))$

As given in slides of lecture-7

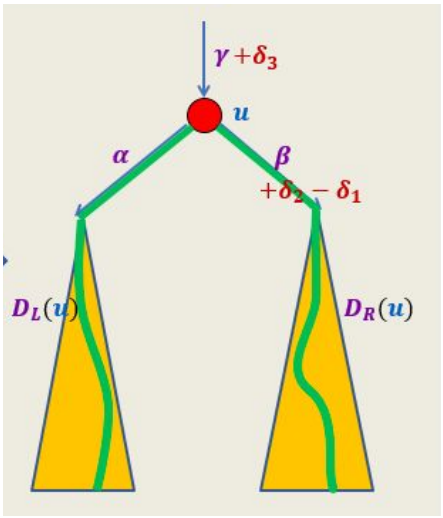
Let's suppose $D_L(u) >= D_R(u)$ and $\delta_1 > 0$ be an optimal solution



Now since node u has to be synchronized then

$$D_L(u) + \delta_1 = D_R(u) + \delta_2$$

But in this case this is not the optimal solution



As we can reduce the total delay enhancement by $\delta_1 > 0$. This is a contradiction.

Hence optimal solution will only be when delay enhancement by u = $|D_R(u) - D_L(u)|$.

(c) int ans=0;

Sync(u)

{if(u is leaf node)return 0;

Else{

D_L <- delay(u,left(u))+Sync(left(u));

D_R <- delay(u,right(u)) + Sync(right(u));

if($D_L > D_R$){

delay(u, right(u)) delay(u, right(u)) + $D_L - D_R$;

return D_L ;

}

Else{

delay(u, left(u)) delay(u, left(u)) + $D_R - D_L$;

return D_R;

}

ans = ans + $|D_L - D_R|$;

}}