SE3352: Algorithm Design December 15, 2021

Notes –Link/Cut Trees

Instructor: *Guoqiang Li* Scribes: *ZhongYi Feng*

1 Introduction

- Link/Cut Trees were developed by Sleator and Tarjan[1][2].They achieve logarithmic amortized cost per operation for all operations.
- It is a data structure solving Dynamic Tree Problem.
- The core idea of Link-Cut trees is to divide a tree into several Splays, and maintain the original tree by operating the Splays.

2 Problem Description

2.1 Definition of Dynamic Tree Problem

Link-Cut Tree is used to sovle **Dynamic Tree Problem**, which supports the following operations:

- Modify the weight of the path between two points
- Query the weight sum of the path between two points
- Query subtree weight sum
- Modify subtree weight
- Disconnect and connect some edges, make sure it is still a tree

2.2 Definition of Link-Cut Trees

Link-Cut Trees represent each tree T in the forest as a tree of auxiliary trees, one auxiliary tree for each preferred path in T.

2.3 Definition of Auxiliary Trees

Auxiliary Trees are splay trees with each node keyed by its depth in its represented tree.
For each node v in its auxiliary tree all the element in its right subtree are higher than v in *v's* represented tree.

2.4 Definition of Represented Tree

The abstract trees that the data structure represents is **represented tree**.

2.5 Definition of Preferred Path

A **preferred path** is a maximal continuous path of **preferred edges** in a tree, or a single node if there is no preferred edges incident on it.
A **preferred edge** is an edge between a **preferred child** and its parent.
The **preferred child** of node v is equal to its *i-th* child if the last access within *v's* subtree was to v itself or if there were no accesses to *v's* subtree at all.

2.6 Definition of Access

If a vertex is passed to any of the operations from above as an argument, the vertex is accessed.

2.7 Definition of Path-Parent Pointer

Auxiliary trees are joined together using **path-parent pointer**.There exists one path-parent pointer per auxiliary tree and it's stored in the root of the auxiliary tree.

3 Algorithms/Models

3.1 Operations of Link-Cut Tree

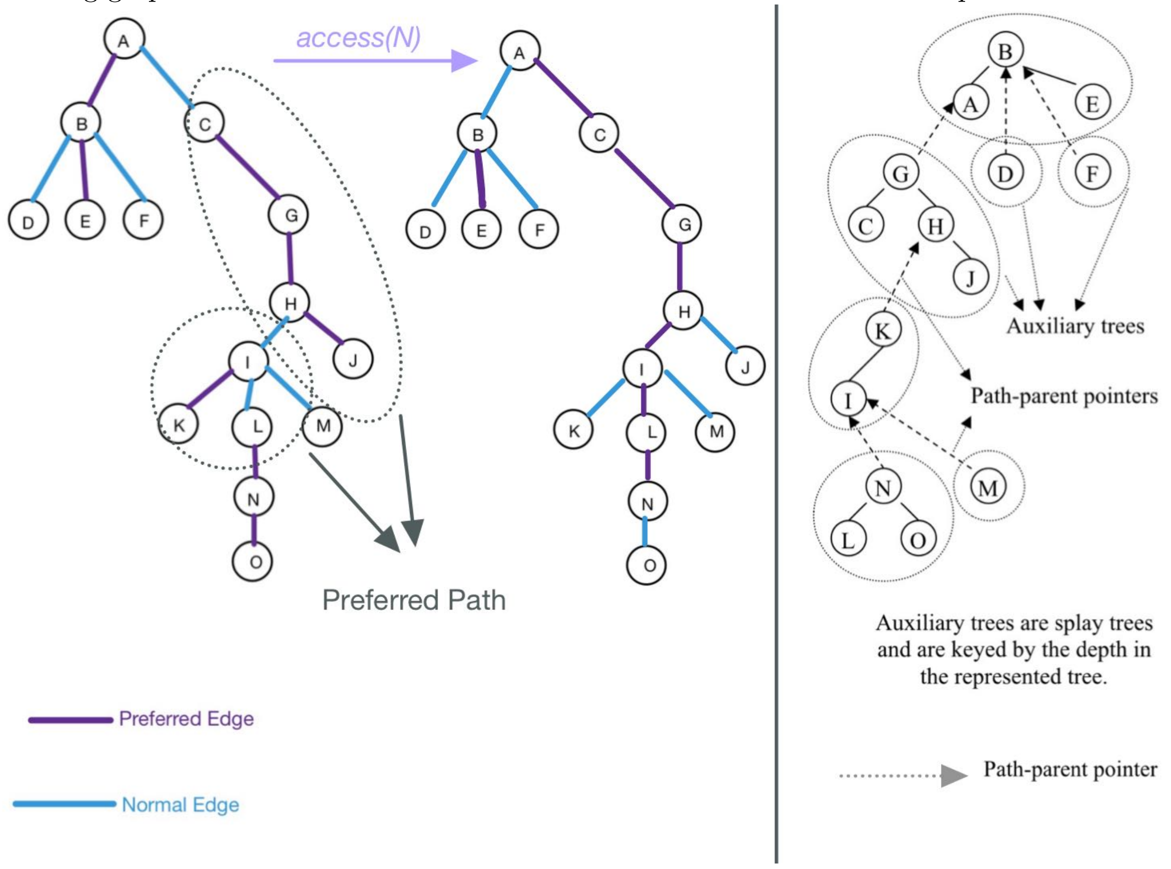
Link-Cut Trees support operations in $O(\lg n)$ as follows:

- *make_tree()* – Create a new tree with only one vertex.
- *access(v)* – The most important operation of Link-Cut Tree. Make the path from v to the root vertex into a new preferred path.
- *cut(v)* – Delete the edge between and its parent. *parent(v)* returns *v's* parent where v isn't root.
- *link(v,w)* – Add an edge(v,w) to make vertex v a new child of vertex w. This operation assumes that v is the root of its tree and v and w are nodes of distinct trees.
- *splay(v)* – The splay operation in Splay Tree, which make vertex v rotate to root.
- *find_root(v)* – Returns the root of the tree where v is located. The operation can be used to determine whether two nodes are connected.
- *rotate(v)* – The rotate operation in Splay Tree, which make vertex v rotate to its parent.
- *get(v)* – Judge whether v is the right of its parent

3.2 Access

Assuming that *access(v)* is called, then the path from v to root is a new *preferred path*. If a vertex u that is passed by during the path is not his parent's *preferred child* (u isn't *parent(u)'s* preferred child), then the *preferred path* including *parent(u)* at first will not include *parent(u)* and *parent(u)'s* above part anymore because *parent(u)'s* preferred child will be u by calling *access(v)*.

Following graph is the structure of Link-Cut Tree and illustration of access operation.



The left part of graph is the ***represented tree***. The right part is the ***auxiliary tree*** representation of represent tree.
Let's discuss *access(v)'s* process in detail:
Because the nodes in auxiliary trees are keyed by their depth in its represented tree R, nodes to the left of v are higher than v and nodes to the right are lower. Thus when we access v, its preferred child becomes *null*. So if before the access v was in the middle of a preferred path, after the access the lower part of this path becomes a separated path. This means we have to separate all the nodes less than v in a separate auxiliary tree. The easiest way to do this is to splay on v. For example, we can bring it to the root and then disconnected it right subtree, making it a separate auxiliary tree.
After dealing with *v's* descendants, we have to make a preferred path from v up to the root of represent tree R. After splaying, v is the root and hence has a path-parent pointer to its parent in R, call it w. We need to connect *v's* preferred path with the w the real parent of v, not just path-parent. In other words, we need to set *w's* preferred child to v.
To do this, we have two stages:

- Disconnect the lower part of *w's* preferred path the same way we did for v (splay on w and disconnected its right subtree).
- Connect *v's* auxiliary tree to *w's*.

Since all nodes in *v's* auxiliary tree are lower than any node in *w's*, we have to make v auxiliary tree the

subtree of w. To do this, we can simply disconnect w with its right child, and change the pointer to point v.
Finally, we have to do a second splay of v. Since v is a child of the root w, splaying simply means rotating v to the root.
We continue bulding up the preferred path in the same way until we reach the root of represented tree R. v will have no right child in the tree of auxiliary tree. Following is the pseudo code of the algorithm.

access(v):

```
• splay(v). This will make left subtree contains all the elements highter than v and right subtree
  contains all elements lower than v.

• Remove v's preferred child:
    – $path - parent(right(v)) \longleftarrow v$
    – $right\,(v) \longleftarrow null$

• loop until we reach the root:
    – $w \longleftarrow path - parent\,(v)$
    – splay(w)
    – switch w's preferred child
        ∗ $path - parent\,(right\,(w)) \longleftarrow w$
        ∗ $right\,(w) \longleftarrow v$
        ∗ $path - parent\,(v) \longleftarrow null$
    – $v \longleftarrow w$

• splay(v) for convenience
```

3.3 Cut

cut(v) operation has three steps:

1. Access v.

2. Make v rotate to its auxiliary tree's root.

3. Cut the connection between v and its left tree in its auxiliary tree.

The pseudo code are following.

cut(v) :

```
• access(v)
• splay(v)
• $left\,(v) \longleftarrow null$
```

3.4 Link

Link operation also has three operation:

1. Access *v*.
2. Make the path parent of *v's* auxiliary tree *w*.
3. Access *v*.

The pseudo code are following.

```
link(v,w):
    • access(v)
    • $path - parent\,(v) \longleftarrow w$
    • access(v)
```

3.5 Splay

The pseudo code are following.

```
splay(v):
    • loop until reach v's root
        – $w \longleftarrow path - parent(v)$
        – if $w$ isn't root
            ∗ if get(v) $\neq$ get(w), then $w$=$v$
            ∗ rotate(w)
        – rotate(v)
        – $w \longleftarrow path - parent(v)$
```

3.6 Get

get(v) is a function for splay operation. If *v* is the right of its parent, *get(v)* will return true; else return false.
The pseudo code are following.

```
get(v):
    • $w \longleftarrow path - parent(v)$
    • if v==right(w), return true
    • return false
```

3.7 Rotate

rotate(v) is also a function used by splay operation, which will make v rotate to its parent. If *v's* parent isn't root, *v's* parent's (assuming the vertex is w) parent will remove w and change w to v. And there are five steps:

- Make *v's* child u the child of w.
- Make w the parent of u.
- Make w the child of v.
- Make v the parent of w.
- Make *w's* parent the parent of v.

The pseudo code are following.

rotate(v):

```
• $w \longleftarrow path - parent(v)$
• if get(v): $u \longleftarrow right\,(v)$
• else: $u \longleftarrow left\,(v)$
• if $w$ isn't root:
    – if get(w),$right\,(path - parent\,(w)) \longleftarrow v$
    – else $left\,(path - parent\,(w)) \longleftarrow v$
• if get(v), $right\,(w) \longleftarrow u$;
  else $left\,(w) \longleftarrow u$
• $path - parent\,(u) \longleftarrow w$
• if get(v), $right\,(v) \longleftarrow w$;
  else $left\,(v) \longleftarrow w$
• $path - parent\,(w) \longleftarrow v$
• $path - parent\,(v) \longleftarrow path - parent\,(w)$
```

3.8 Find Root

After *access(v)*, the root node must be the smallest node of the Auxiliary Tree to which v belongs. We first rotate v to the root of the Auxiliary Tree to which it belongs. Then start from v and walk left along the Auxiliary Tree until we cannot go to the left, this point is the root node we are looking for. Since we are using the Splay Tree data structure to save the Auxiliary Tree, we also need to perform Splay operations on the root node.
The pseudo code are following.

find_root(v):

```
• access(v)
• splay(v)
• while left(v) $\neq$ null do $v \longleftarrow left(v)$
• splay(v)
• return $v$
```

4 Practical Algorithm

In the process of a single rotation, not only must the local form of the splay tree be modified, but also the maxcost attribute of the vertex must be maintained. At the same time, because of the usual writing of splay tree, it is necessary to check whether the node is the root node or the father. This also took a lot of time. Thus there exists a concise and efficient way to write SPLAY, ROTATE, UPDATE as follows:

```
procedure UPDATE(node)
    • $k \longleftarrow cost[node]$
    • $x \longleftarrow maxcost[child[node][0]]$
    • $y \longleftarrow maxcost[child[node][1]]$
    • if $x < y$ then
        – $x \longleftarrow y$
    • end if
    • if $k < x$ then
        – $k \longleftarrow x$
    • end if
    • $maxcost[node] \longleftarrow k$
end procedure
```

```
procedure ROTATE(node,x)
    • $p \longleftarrow parent[node]$
    • $y \longleftarrow nodetype[p]$
    • $tmp \longleftarrow parent\,[p]$
    • $parent\,[node] \longleftarrow tmp$
    • $nodetype\,[node] \longleftarrow y$
    • **if** y $\neq$ 2 **then**
        – $child\,[tmp]\,[y] \longleftarrow node$
    • **end if**
    • $y \longleftarrow 1 - x$
    • $tmp \longleftarrow child\,[node]\,[y]$
    • $child\,[p]\,[x] \longleftarrow tmp$
    • $parent\,[tmp] \longleftarrow p$
    • $nodetype\,[tmp] \longleftarrow x$
    • $parent\,[p] \longleftarrow node$
    • $nodetype\,[p] \longleftarrow y$
    • $child\,[node]\,[y] \longleftarrow p$
    • **UPDATE(p)**
**end procedure**

**procedure** SPLAY($node$)
    • **repeat**
        – $a \longleftarrow nodetype\,[node]$
        – **if** a = 2 **then**
            ∗ break
        – **end if**
        – $p \longleftarrow parent\,[node]$
        – $b \longleftarrow nodetype\,[p]$
        – **if** a = b **then**
            ∗ ROTATE($p$,$a$)
        – **else**
            ∗ ROTATE($node$,$a$)
        – **end if**
        – if b=2 then
            ∗ break
        – end if
        – ROTATE(node,b)
    • until FALSE
    • UPDATE(node)
end procedure
```

In the process of SPLAY (*node*), the usual wording of SPLAY will perform many times of UPDATE on some points. In fact, this is unnecessary. We only need to update the parent node of the rotated node in the ROTATE operation. And at the end of the SPLAY, perform an UPDATE on the node. Since UPDATE is a time-consuming operation, this optimization removes about half of the redundant UPDATE operations, so the performance of the program has been greatly improved.

5 Key properties

It can easily find the amortized time complexity is at most $O(\log n)$ for all operations except *ACCESS*. In the analysis of *ACCESS* operation, we divide the *ACCESS* time into two parts. In the first part, we prove that the time of switching *Preferred Child* is amortized $O(\log n)$. In the second part we prove that the total time of all Splay operations in *ACCESS* is amortized $O(\log n)$.

5.1 Heavy-Light Decomposition

Heavy-Light Decomposition is an analytical technique applicable to any tree. It divides the edges of the tree into two types, one of which is either Heavy or Light.
Let *size(i)* be the number of nodes in the tree whose root is i and u be the largest son of *size(u)* of *v's* sons. We call the edge *(v, u)* heavy and the edge *(v, w)* from v to its other son w light. If a node has sons, there is a single heavy edge that starts from it. According to this definition, it can be obtained directly:
Property 3.1 If u is a son of v and *(v, u)* is a light edge, then

$$size(u) < size(v)/2$$

Proof:

- Assuming $size(u) \geq size(v)/2$, since *(v, u)* is a light edge, there exists a heavy edge *(v, w)* from v. Depending on the heavy edge definition,$size(v) \geq 1+size(w)+size(u) \geq 2size(u)+1 \geq 1+size(v)$. Which is Contradiction.

Let *light-depth(v)* be the number of light edges passing from v to the root, then
Lemma 1

$$light - depth(v) \leq logn$$

Proof:

- According to property 3.1, if a light edge is passed, the number of nodes in the current sub-tree is at most half of the original one. The number of nodes in the original sub-tree is n, and the sub-tree of v is at least 1, so it has passed $\log n$ light edges at most.

5.2 Proof of the number of changes of Preferred Child

In order to calculate the number of changes of the Preferred Child, we first perform Heavy-Light Decomposition on the tree T.
Except the disappearance of the Preferred Child of v, every change of the Preferred Child will inevitably generate a new Preferred Edge. So We just need to count the number of newly created Preferred Edges. According to Lemma 3.1, at most $\log n$ light Preferred Edge will be generated during *ACCESS* operation.We know the number of times each heavy edge becomes Preferred Edge is at most the number of times the heavy edge changes from Preferred Edge to normal edge plus the total number of edges.And **a heavy edge changes from Preferred Edge to normal edge must accompany with a light edge changes to Preferred Edge**, so the the number of times each heavy edge becomes Preferred Edge is at most the number of times the light edge changes to Preferred Edge plus the total number of edges. So, on average, the number of times that the heavy edges in each *ACCESS* operation become Preferred Edge is $O(\log n)$.
To sum up, the number of changes of Preferred Child are evenly amortized $O(\log n)$..

5.3 Proof of the amortization of the total time of Splay operation $O(\lg n)$

We commonly use potential energy analysis method for the time complexity analysis of Splay operation. Assign a positive weight *w(u)* to each node u in the Splay Tree, and let *s(u)* be the sum of the weights of the nodes in the subtree of u.Define the potential energy function $\Phi=\sum_u \lg s(u)$.
Assuming c_i is the actual cost of i-th operation, then we can describe the average cost of i-th operation as $c_i + \Phi_i - \Phi_{i-1}$. Then the average cost of n operations:

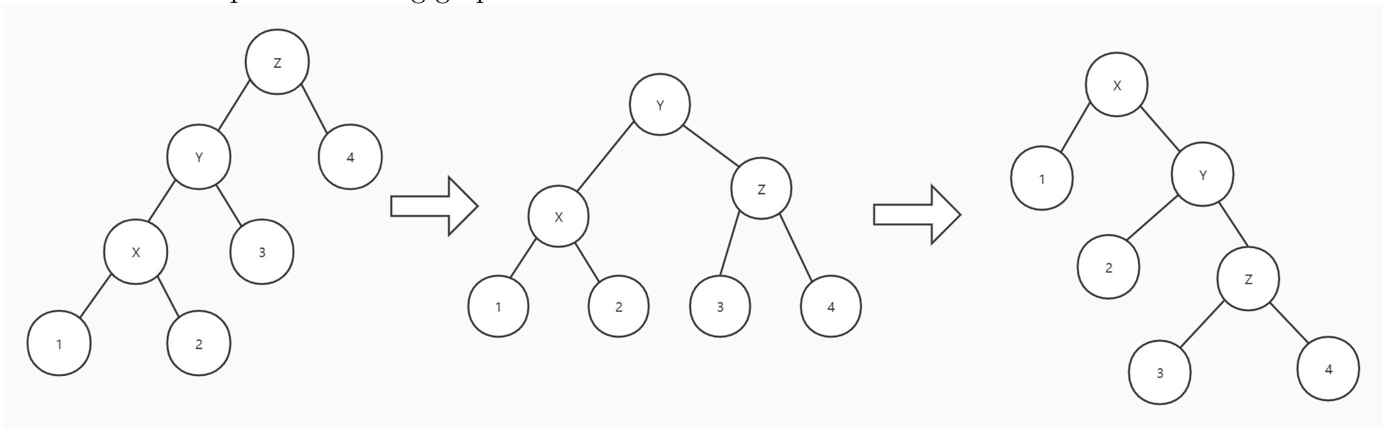
$$a_n = \sum_{i=1}^{n} (c_i + \Phi_i - \Phi_{i-1}) = \sum_{i=1}^{n} c_i + \Phi_n - \Phi_0$$

Let $\Phi_0 = 0$,then $a_n = \sum_{i=1}^{n} c_i + \Phi_n \geq \overline{cost}$
Let $\mu(x) = \log_2 s(x)$. Then $\Phi_i(x) = \sum_{i=1}^{n} \mu_i(x)$
Lemma 2 zig-zig, zag-zag average time spent in operation *Splay(v)*

$$\overline{cost} \leq 3(\lg s'(v) - \lg s(v))$$

Proof:

- Let's see an example as following graph:



- $a_n = 2 + \mu(x') + \mu(y') + \mu(z') - \mu(x) - \mu(y) - \mu(z)$
- It's obvious that $\mu(x') = \mu(z)$
- Thus $a_n = 2 + \mu(y') + \mu(z') - \mu(x) - \mu(y)$
- It's obvious that $\mu(y) > \mu(x)$ and $\mu(x') \geq \mu(y')$
- Thus $a_n \leq \mu(y') + \mu(z') - 2\mu(x) + 2 \leq \mu(x') + \mu(z') - 2\mu(x) + 2$
- $s(x) + s(z') = 2 + s(1) + s(2) + s(3) + s(4) \leq s(x')$
- Thus $s(x) \cdot s(z') \leq \frac{1}{4} s^2(x')$
- Thus $\mu(x) + \mu(z') - 2\mu(x') \leq -2$
- That's to say,$-2 - \mu(x) - \mu(z') + 2\mu(x') \geq 0$
- Add it to a_n, we can get

$$a_n \leq 3\mu(x') - 3\mu(x)$$

- Thus, we can prove that

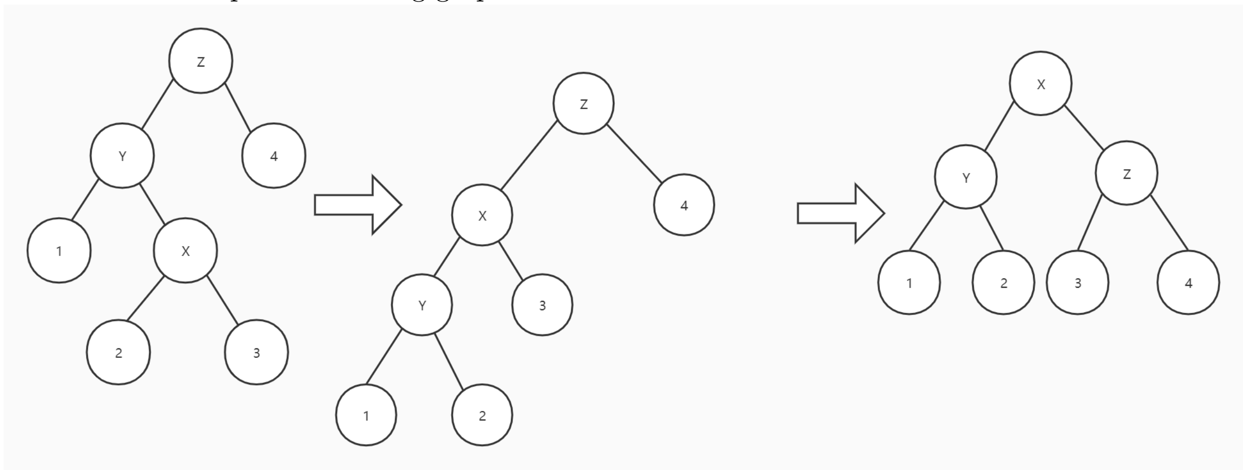
$$\overline{cost} \leq 3(\lg s'(v) - \lg s(v))$$

Lemma 3 zig-zag, zag-zig average time spent in operation *Splay(v)*

$$\overline{cost} \leq 2(\lg s'(v) - \lg s(v)) \leq 3(\lg s'(v) - \lg s(v))$$

Proof:

- Let's see an example as following graph:



- $a_n = 2 + \mu(x') + \mu(y') + \mu(z') - \mu(x) - \mu(y) - \mu(z)$
- Because of $\mu(x') = \mu(z)$

- $a_n = 2 + \mu(y') + \mu(z') - \mu(x) - \mu(y)$
- Similar to Lemma 2, we can get $a_n \leq 2 + \mu(y') + \mu(z') - 2\mu(x)$
- Similar to Lemma 2, we can get $-2 - \mu(y') - \mu(z') + 2\mu(x') \geq 0$
- Thus

$$a_n \leq 2\mu(x') - 2\mu(x) \leq 3\mu(x') - 3\mu(x)$$

- Thus, we can prove that

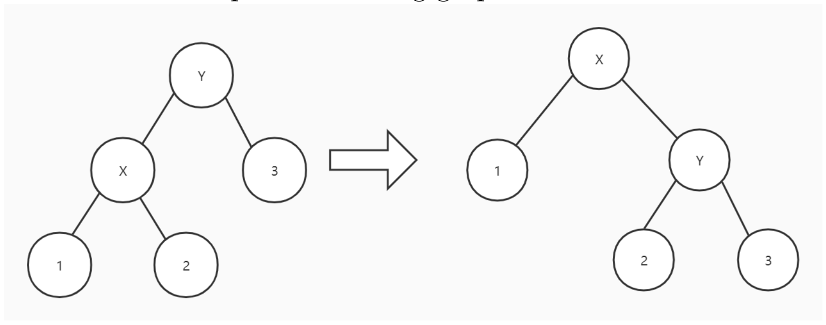
$$\overline{cost} \leq 2(\lg s'(v) - \lg s(v)) \leq 3(\lg s'(v) - \lg s(v))$$

Lemma 4 zig, zag average time spent in operation *Splay(v)*

$$\overline{cost} \leq \lg s'(v) - \lg s(v) + 1 \leq 3(\lg s'(v) - \lg s(v)) + 1$$

Proof:

- Let's see an example as following graph:



- $a_n = 1 + \mu(x') + \mu(y') - \mu(x) - \mu(y)$
- Because $\mu(x') = \mu(y)$ and $\mu(x') \geq \mu(y')$
- We can get

$$a_n = 1 + \mu(y') - \mu(x) \leq 1 + \mu(x') - \mu(x) \leq 1 + 3\mu(x') - 3\mu(x)$$

- Thus

$$\overline{cost} \leq \lg s'(v) - \lg s(v) + 1 \leq 3(\lg s'(v) - \lg s(v)) + 1$$

Lemma 5 (Access Theorem) The average time spent *Splay(v)*

$$\overline{cost} \leq 3(\lg s'(v) - \lg s(v)) + 1$$

Proof:
This can be proved easily by Lemma2, Lemma 3 and Lemma4.

In this problem, since an *ACCESS* operation will perform multiple (equalized $O(\log n)$ times) *SPLAY* operations, we should find a suitable weight function instead of simply let *w(u)* = 1. Let's imagine the edges in Auxiliary Tree as *real edges*, and think of Path Parent is imagined as a *virtual edge.*The total number of nodes of a subtree is equal to 1 add the total number of nodes of all its subtrees (including the subtrees connected by the virtual edge). Let *w(u)* equal to 1 add the total number of nodes in the subtree with u as the Path Parent, then *s(u)* is the total number of nodes in the subtree of u.
Assuming in the process of *ACCESS(v)*, we respectively enter $v_0 = v$, $v_1 = path - parent\,[v_0]$,..., $v_k = path - parent\,[v_{k-1}]$ to *SPLAY* operation. Then, the average time of *ACCESS* is

$$\begin{gathered}\overline{cost} \le \textstyle\sum_{i=0}^{k} 3\left(\lg s'\left(v_i\right) - \lg s\left(v_i\right)\right) + 1 \\ \le 3\left[\textstyle\sum_{i=1}^{k}\left(\lg s'\left(v_i\right) - \lg s'\left(v_{i-1}\right)\right) + \lg s'\left(v_0\right) - \lg s\left(v_0\right)\right] + k \\ = 3\left(\lg s'\left(v_k\right) - \lg s\left(v_0\right)\right) + k \\ \le 3\lg n + \text{ the number of changes of Preferred Child}\end{gathered}$$

In 5.2, we have proved that the number of changes of Preferred Child equalized $O(\log n)$ times, so the amortized time complexity of *ACCESS* is $O(\log n)$.

6 Conclusion

Link-Cut Tree only adds a few lines of code on the basis of the existing *Splay*, which can perfectly solve the problems related to *Dynamic Trees.* With the powerful tool of Link-Cut Tree, many tree or ring-plus extroverted trees, and even dynamic problems on cacti can be easily solved.
In *ACCESS(v)*, if there are only normal edge, the operation will jump from vertex v to root, which will take $O(n)$. But the time complexity of Link-Cut Tree is $O(n \lg n)$, which was proved by Tarjan.

7 References

References

[1] D. D. Sleator, R. E. Targan, *A Data Structure for Dynamic Trees*, Jornal. Comput. Syst. Sci., 28(3):362-391, 1983.

[2] R. E. Tarjan, *Data Structure and Network Algorithms*, Society for Industrial and Applied Mathmatics, 1984.