Dynamic Connectivity

Project Final Report

Charbel Chucri, Jihad Hanna, Hadi Kotaich

Professor Amer Mouawad

CMPS 396AC – Fall 2020

Contents

I. Overview

Dynamic connectivity structures aim to solve problems related to queries on graphs and trees that involves addition and removal of edges. In our project we focused on trees mainly and tackled a special case on graphs (offline queries). In this project we first tried to understand how it is solved on trees, we then implemented two solutions namely the link cut tree which was invented by Sleator and Tarjan and the Euler tour tree which was invented by Henzinger and King. We tested correctness on online judges and then tested speed locally. Once we were done with trees, we tackled the problem on graphs and implemented a solution with the restriction that queries are offline. We were motivated to work on this problem due to many reasons: first it is very useful in competitive programming to know about such data structures. Second although you might find some implementations online, understanding the algorithm is crucial to use it and be able to change it according to the given problem. Third what we did will improve the prewritten code repertoire we have as AUB competitive programmers and it will hopefully be useful for future generations. This is why it was important for us to implement and test well what we wrote and not only understand the algorithm.

II. Formal Problem Statement

We are given an **acyclic** undirected graph G = (V, E) and multiple updates/queries that we should maintain/answer:

- cut(u, v): Assuming there is at this point an edge between nodes u & v, remove that edge.
- link(u, v): Assuming there is at this point no **~~edge~~ path** between nodes u & v, add an edge between u & v.
- isConnected(u, v): Check if nodes u & v are in the same connected component, i.e. check if there's currently a path from u to v.

We made two relaxations to the general problem (shown in bold) to make sure that throughout all computation G is an undirected forest.

The problem can be easily solved in O(1) update time and O(|V| + |E|) query time using BFS/DFS but in our work we're aiming for better, namely something in the order of O(log n) for both queries and updates.

III. First Method: Euler Tour Trees

Our first approach relies on a data structure called Euler Tour Trees, and throughout the rest we assume the reader is already familiar with the concept of Euler tours.

1. Prerequisite: Balanced Binary Search Tree (Treap)

In order to maintain our data structure, we will need a balanced binary search tree that supports the following operations:

- insert(v, idx): inserts value v at index idx (which means that we assume v to be the idx's smallest element in the BST)
- delete(idx): deletes the idx's smallest element from the BST
- merge(T_1,T_2): merges two BST's into one assuming the keys in T_1 are less than all the keys in T_2.
- split(T, idx): splits the BST into 2 BST's, one containing the smallest idx elements and the other containing the rest.
- findIdx(v): given a node in the BST, returns the number of smaller values.

All operations should preserve the balance of the BST. In our implementation we use the Implicit Treap data structure that achieves all aforementioned operations in O(log n) expected time. We will also augment this data structure so that each node contains a pointer to its parent in the Treap as this will turn out to be useful later on in our implementation.

2. Building the Data structure

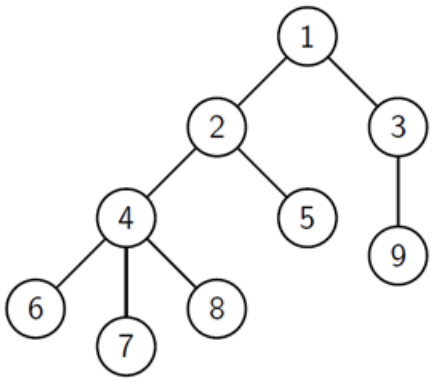
Each component in our forest will be a tree, and for each such component we will store its Euler Tour in a treap the following way:

- We root our tree at an arbitrary node. (We will be able to change this root later on…)
- We traverse the tree in DFS-order and we store each edge in the Treap 2 times, once when going down to the children's subtree and once when leaving it back to the parent.
- We also store in the Treap a loop edge (u, u) for each node visited for the first time.
- We store for each node a pointer to its loop edge, and for each edge pointers to its two occurrences in the Euler Tour.

An edge's key in the Treap is its order of appearance in the Euler Tour. Below you can see our C++ implementation: (When not given an index in insert, Treap inserts at the end)

```
vector<TreapNode*> self; int n;
unordered_map<pair<int, int>, pair<TreapNode*, TreapNode*>> edges;

EulerTourTree(Forest F) : n(F.n), self(F.n) {
    vector<bool> vis(n, false);
    for (int i = 0; i < n; i++) if (!vis[i]) {
        TreapNode* root = NULL; eulerTour(F, root, vis, i);
    }
}

void eulerTour(Forest& F, TreapNode*& root, vector<bool>& vis, int cur, int par = -1) {
    TreapNode* curN = new TreapNode(cur, cur);
    treapInsert(root, curN); self[cur] = curN;
    vis[cur] = true;

    for (int& e : F.adj[cur]) if(e != par) {
        TreapNode* inE = new TreapNode(cur, e);
        treapInsert(root, inE);
        eulerTour(F, root, vis, e, cur);
        TreapNode* outE = new TreapNode(e, cur);
        treapInsert(root, outE);
        edges[{cur, e}] = edges[{e, cur}] = { inE, outE };
    }
}
```

The following is an example of the Euler Tour we get on a certain tree:



We get the following Euler Tour:

(1, 1) (1, 2) (2, 2) (2, 4) (4, 4)
(4, 6) (6, 6) (6, 4) (4, 7) (7, 7)
(7, 4) (4, 8) (8, 8) (8, 4) (4, 2)
(2, 5) (5, 5) (5, 2) (2, 1) (1, 3)
(3, 3) (3, 9) (9, 9) (9, 3) (3, 1)

The nice thing of such representation is that invariants are very simple:

- Each edge is stored exactly twice — once in each direction. (Except for loops)
- There are exactly $3n - 2$ elements of the tour. (2(n – 1) edges and n loops)
- You can cyclically shift the tour and it remains a tour, with maybe the root changing.
- In particular, first node of the first edge is the same as the second node of the last edge.

3. Reroot

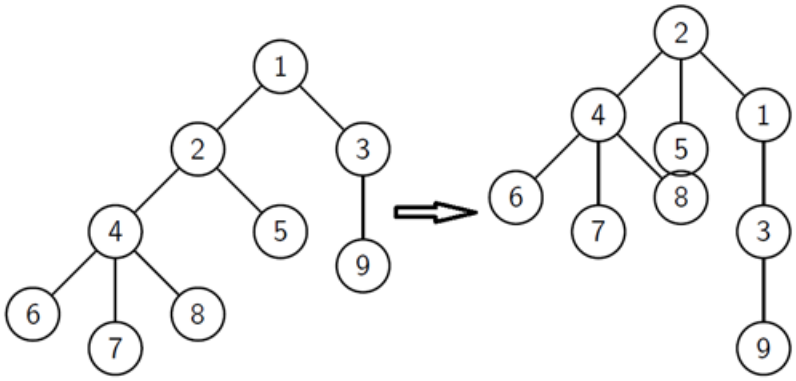
The first operation this Data Structure will support is rerooting and it will turn out to be crucial in implementing the "link" operation. As we mentioned earlier, rerooting an tree corresponds to a cyclical shift of the Euler Tour, so we reroot the following way:

- Locate the loop (newRoot, newRoot) in our Treap and find its index idx in the Treap.
- Split the Treap into 2 parts: Everything before idx and the rest.
- Merge back the 2 parts in reverse order.

Below you can see our C++ implementation:

```
void reroot(int cur) {
    TreapNode* firstCur = self[cur];
    int idx = 0; treapFindIdx(firstCur, idx);
    TreapNode* left = nullptr, * right = nullptr;
    treapSplit(firstCur, left, right, idx - 1);
    TreapNode* root = nullptr;
    treapMerge(root, right, left);
}
```

Below is an example of rerooting



(1, 1) (1, 2) (2, 2) (2, 4) (4, 4)
(4, 6) (6, 6) (6, 4) (4, 7) (7, 7)
(7, 4) (4, 8) (8, 8) (8, 4) (4, 2)
(2, 5) (5, 5) (5, 2) (2, 1) (1, 3)
(3, 3) (3, 9) (9, 9) (9, 3) (3, 1)

(2, 2) (2, 4) (4, 4) (4, 6) (6, 6)
(6, 4) (4, 7) (7, 7) (7, 4) (4, 8)
(8, 8) (8, 4) (4, 2) (2, 5) (5, 5)
(5, 2) (2, 1) (1, 3) (3, 3) (3, 9)
(9, 9) (9, 3) (3, 1) (1, 1) (1, 2)

4. Find Root and IsConnected

FindRoot(u) returns the current root of the component of u and can easily be found the following way that uses the fact that the root is always the first node of the first edge in the Treap:

- Locate the loop (u, u) in the Treap
- Move up using the parent pointers we stored at first until we reach the root of the Treap. (not necessarily of the tree)
- Move to the left-most node in the Treap which represents the smallest element or the first element of the Euler Tour.

Now to check if 2 nodes are in the same connected component, we just need to check if they have the same root. Below is our C++ implementation.

```
TreapNode* findRoot(int cur) {
    TreapNode* curN = self[cur];
    while (curN->p != nullptr) curN = curN->p;
    while (curN->l != nullptr) curN = curN->l;
    return curN;
}

bool connected(int u, int v) {
    return findRoot(u) == findRoot(v);
}
```

5. Cut

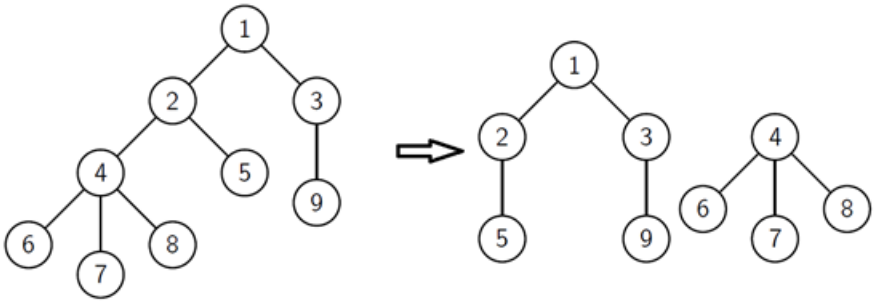
To cut an edge u-v we proceed to following way:

- Locate the 2 occurrences of the uv-edge in the Treap and find their indices.
- Split the Treap 2 times to obtain 3 Treaps:
    1. Left which contains everything before the first occurrence of the uv-edge.
    2. Mid which contains everything between the two uv-edges inclusive.
    3. Right which contains everything after the second occurrence of the uv-edge.
- Remove first and last nodes of mid which are the edge to be cut to get the child's subtree
- Merge Left & Right Treaps to get the parent's Tree.

Below is our C++ Implementation:

```
void cut(int u, int v) {
    assert(edges.count({ u, v }));

    pair<TreapNode*, TreapNode*> p = edges[{u, v}];
    TreapNode* first = p.first, * last = p.second;
    int fID = 0; treapFindIdx(p.first, fID);
    int sID = 0; treapFindIdx(p.second, sID);
    if (fID > sID) { swap(fID, sID); swap(first, last); }

    TreapNode* curLeft = nullptr, * right = nullptr;
    treapSplit(p.first, curLeft, right, sID - 1);
    treapErase(right, 1);


    TreapNode* left = nullptr, * childTree = nullptr;
    treapSplit(curLeft, left, childTree, fID - 1);
    treapErase(childTree, 1);

    TreapNode* root = nullptr;
    treapMerge(root, left, right);

    edges.erase({ u, v }); edges.erase({ v, u });
}
```

Below is an example of a cut operation and the corresponding Euler Tours:



(1, 1) (1, 2) (2, 2) (2, 4) (4, 4)
(4, 6) (6, 6) (6, 4) (4, 7) (7, 7)
(7, 4) (4, 8) (8, 8) (8, 4) (4, 2)
(2, 5) (5, 5) (5, 2) (2, 1) (1, 3)
(3, 3) (3, 9) (9, 9) (9, 3) (3, 1)

(1, 1) (1, 2) (2, 2) (2, 5) (5, 5)
(5, 2) (2, 1) (1, 3) (3, 3) (3, 9)
(9, 9) (9, 3) (3, 1)
and
(4, 4) (4, 6) (6, 6) (6, 4) (4, 7)
(7, 7) (7, 4) (4, 8) (8, 8) (8, 4)

6. Link

To add a uv-edge with v child of u in the Tree (Design decision) we proceed as follows:

- Make sure v is the root of its tree by calling reroot(v)
- Locate the loop (u, u) in the Treap and find its index
- Split the Treap into two, Left containing everything before the loop inclusive and Right containing everything after the loop
- Add (u, v) to the end of Left and (v, u) to the beginning of Right.
- Merge v's Treap with Left and then merge the result with Right in this order.

Below is our C++ implementation:

```
void link(int par, int child) {
    reroot(child); TreapNode* cur = self[par];
    int idx = 0; treapFindIdx(cur, idx);
    TreapNode* left = nullptr, * right = nullptr;
    treapSplit(cur, left, right, idx);

    TreapNode* inE = new TreapNode(par, child);
    TreapNode* outE = new TreapNode(child, par);
    edges[{par, child}] = edges[{child, par}] = { inE, outE };



    treapInsert(left, inE); treapInsert(right, outE, 1);
    TreapNode* root = nullptr, * childRoot = self[child];
    treapFindIdx(childRoot, idx);
    treapMerge(root, left, childRoot);
    treapMerge(root, root, right);
}
```

A link operation example is given by reading the above example backwards.

7. Time Complexity & Testing

All operations mentioned performs a constant number of Insert/Delete/Merge/Split/FindIdx operations and hence all queries/updates are achieved in O(log n) expected time.

This method can be modified to solve the general DCP (Without the relaxations introduced above) in $O(\log^2 n)$ amortized time complexity.

We tested our implementation for correctness on two online judges: SPOJ & Codeforces, with success.

IV. Second Method: Link Cut Trees

We already saw that the Euler Tour Tree solves the dynamic connectivity problem efficiently. We will now see how to user another data structure, namely the Link-Cut Tree, to solve the same problem.

On one hand, we wanted to implement another data structure that also solves the dynamic connectivity problem. Having two different options would allow us to compare their running time. In fact, this study aims at finding which one is more efficient in practice, knowing that both data structures have similar asymptotic running times.

On another hand, the Euler Tour Tree only allows us to link/cut edges and to check whether two nodes are connected. What if we also wanted to find some values along a path (e.g., the distance between two nodes u and v)? We will see how using a Link Cut Tree allows us to solve such problems.

I. Prerequisite: Splay Tree

A) Idea

A splay tree is an implementation of a Balanced Binary Search Tree. It does what a typical BBST can do: adding nodes, removing nodes and searching for nodes given a key in $O(\log n)$ amortized time.

What differs between splay trees and other implementations of BBST is the $splay$ function. This function takes a node u as parameter and, through a series of rotations, it puts u to be the new root of the splay tree. The rotations will maintain the properties of the binary search tree.

This $splay$ function is used whenever we access/add a new node in the BST. This is what makes this BST balanced (on average). The intuition behind why this works is that a splay tree behaves like a cache. The nodes that are the closest to the root are the ones which were most recently accessed. Thus, a subsequent call to one of them will be dealt with much faster.

We will also augment this BBST to be able to find some subtree values as those can be quite useful.

B) Implementation

We have two main classes in our implementation: $SplayNode$ and $SplayTree$.

A $SplayNode$ will have 2 children (left and right child in the BBST), a value (since this is a BST) and a subtree size (to get useful subtree values). It will also have a pointer to its parent.

A $SplayNode$ has 3 main functions (the others are for convenience):

i) $Rotate(\,)$: The goal of rotation is to reduce the depth of the node by 1. We basically swap it with its parent, while maintaining their children so that the values respect the BST properties. Rotation is done in constant time.
ii) $Splay(\,)$: We keep rotating the node until it becomes the root of the tree. Splaying is done in $O(h)$ time where h is the height of the tree which is logarithmic if we consider the amortized cost.
iii) $Update(\,)$: We update the subtree size according to the children (that might have been changed). It is called whenever some changes occur with the tree structure. $Update$ is done in constant time.

A $SplayTree$ is defined by its root (a $SplayNode$). It has 4 main functions (note that the other functions were written during testing/problem solving and will be explained in the appropriate section):

i) $Splay(SplayNode * node)$: It calls the splay function on the node and then sets it as the new root of the tree.
ii) $Insert(int\ x)$: It creates a new $SplayNode$ with value x and inserts it in the tree. The insertion is typical for a BST, we just go left/right according to the value. Note that we splay the added node in the end.
iii) $Erase(SplayNode *\ node)$: We first splay $node$ to get it as the root. Now, the idea is to join the left and right subtree without that $node$. We use the $join(SplayNode * a, SplayNode * b)$ auxiliary function to do so. It basically sets b to be the right child of the node with max value in the subtree of a (since every node in the subtree of a has a smaller value than every node in the subtree of b).

All 3 of these operation are done in amortized $O(\log n)$ time due to the tree being balanced on average.

C) Testing and Problem Solving

To test the correctness of our implementation and also to discover some of its applications, we used the Splay Tree to solve problems on online judges in addition to doing some local testing.

We first implemented the $inorder()$ function which just does an in-order traversal and stores the value in an array. This was used for local testing; we just inserted a million nodes with random values and made sure the $inorder()$ function returned these values in sorted order.

We then implemented the $kth(int\ k)$ function. It finds the node with the kth smallest value in the tree. It uses the subtree size augmentation which we maintained using the $update()$ function. To test it, we submitted our code on two different online judges: SPOJ and e-olymp.

II. Definitions

We will first assume that each tree in the input forest is rooted at an arbitrary node. This will allow us to use the concept of children and parents.

Each node will have at most one **preferred child**. If it has no preferred child, we assume a node is its own preferred child for convenience.

We can now define a **preferred path** to be a maximal path $u_1, u_2, \dots, u_k$ such that u_{i+1} is the preferred child of u_i for $1 \leq i \leq k-1$. Note that since the path is maximal, u_k is its own preferred child and u_1 is not the preferred child of any node. We can have preferred paths of length 1.

III. Target API

The link-cut tree will have functions similar to the Euler Tour tree to add/remove edges and to check connectivity. In addition to that, a link-cut tree will be able to find some path value from the root to a node. We will later explain how to generalize this to any path.

So, the main functions of the Link-Cut Tree will be:

i) $Link(u, v)$: Adds an edge between u and v
ii) $Cut(u)$: Removes the edge connecting u with its parent
iii) $FindRoot(u)$: Finds the root of the tree that contains u
iv) $PathAggregate(u)$: Finds some value for the path from the root to u

We will also need some helper functions, mainly:

i) $Access(u)$: Makes the path from the root to u a preferred path
ii) $LCA(u, v)$: Finds the Least Common Ancestor (LCA) of u and v

Note that each function should have an amortized running time of $O(\log n)$.

IV. Initial Implementation

In our implementation, we do not keep track of the input tree as is. Instead, we represent each preferred path by a splay tree where the key of each node is its depth in the input tree. The figure below shows such an example. An edge uv is in bold if u is the parent of v and v is the preferred child of u. If we consider the preferred path a, b, d, we can see that since a has depth 0, b has depth 1 and d has depth 2, a is the leftmost node and d is the rightmost node in the corresponding splay tree.

The edges in bold can be reconstructed using the splay tree edges. We still need to keep track of the other edges. We will call those **path parent pointers** since they are edges that link 2 preferred paths. Each preferred path (and thus splay tree) will have one path parent pointer that points to the path just above it in the input tree (thus to another splay tree). Those edges are the dotted ones in the splay tree representation.

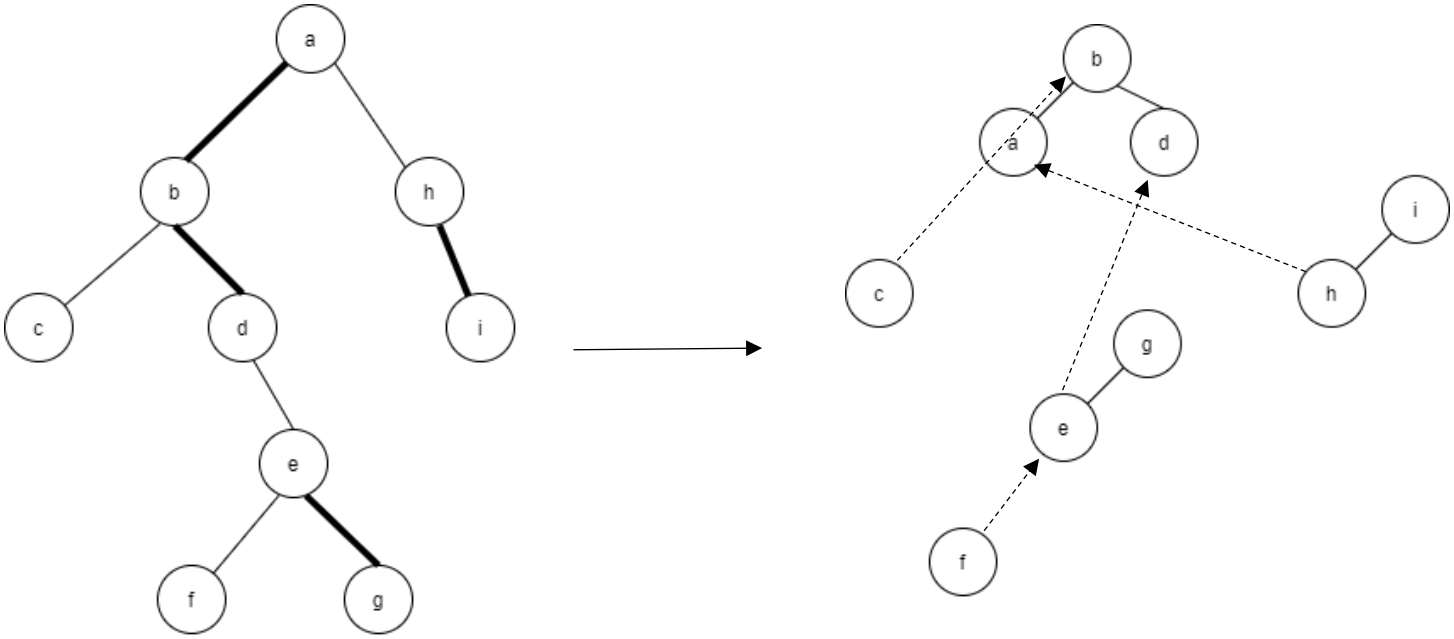


Figure 1: Representation of a tree using splay trees

Hence, we keep track of the splay trees and the preferred path pointer for each one of those. We mentioned that the key of a node in a splay tree is its depth in the input tree. It is important to point out that this key is **implicit**. We do not store it with each node since otherwise, once we link 2 nodes, we will have to

update the depth of up to $O(n)$ nodes. This would clearly blow up our target complexity. By keeping the keys implicit, we just have to make sure that the binary search properties are maintained when we add an edge.

In our implementation, we have 3 main classes: $NodeVal, Node$ and $LinkCutTree$.

Note that we do not have a splay tree class for the same reason mentioned above. Our tree being dynamic, we might need to merge 2 splay trees and having such classes would make it messier implementation-wise. Instead, we only keep track of the splay tree nodes. Those are enough since we can just call $splay$ on a node u to make it the root of its splay tree.

The $Node$ is our splay tree node. As such, it has the same components that a splay tree node (i.e., $child$ and $parent$) minus the key (since it is implicit). It also has a $NodeVal$ where we store subtree values (and update them). The idea is that we only have to modify $NodeVal$ later on to add new properties to our Link-Cut Tree. In addition to those, a $Node$ has a path parent pointer and a $reverse$ boolean. The use of the latter will be explained later.

Regarding the $Node$'s functions, we modified the $rotate()$ function so that the path parent pointer is kept at the root of each splay tree. We also have a new $applyReverseLazy()$ function which will be explained later.

The $LinkCutTree$ stores a vector of nodes and its size n. The user will then be able to specify a node by its index rather than by its pointer. We will now explain in detail how each function works.

$AddNode(NodeVal)$ simply creates a new node (which will be a preferred path on its own), add it to the vector of nodes and return its id.

```
int addNode(NodeVal* nodeVal) {
        nodes.push_back(new Node(n, nodeVal));
        return n++;
}
```

$Access(u)$ makes the path from the root to u a preferred path.

To implement it, we start by splaying u, making it the root of the splay tree that contains it. Then we detach it from its right child. The idea here is that the right subtree of u consists of nodes with bigger depth in the represented tree. Since we want a preferred path to end at u, u shouldn't have any preferred child. Hence, we detach it from its right subtree.

```
Node* access(Node* u) {
        u->splay();
        u->detachChild(1);

        Node* curPP = u;
        while (u->pathParentPointer) {
                curPP = u->pathParentPointer;
                curPP->splay();
                curPP->detachChild(1);
                curPP->attach(u, 1);
                u->pathParentPointer = nullptr;
                u->splay();
        }

        return curPP;
}
```

Then, we go to the path parent of u and detach it from its descendants in the represented tree (i.e., its right subtree in the splay tree) since its preferred child will change. Now, we link it to u. We basically linked the previous preferred path to the one that contains u. We splay u again so that it becomes the root of the splay tree. We repeat this process until we get to the root. We now have a preferred path from the root to u.

We return the last path parent pointer used. This return value will be useful in the $LCA(u, v)$ function.

The amortized time complexity is $O(\log n)$. Rather than including a proof in this report (which is found in Tarjan's paper), we will show this is accurate by testing the running time of our data structure.

$Link(u, v)$ adds an edge between u and v. It assumes u and v are not connected. It also assumes that u ($child$ in the code) is a root in the represented tree. We will later discuss how to deal with this assumption.

```
void link(Node* child, Node* parent) {
        assert(findRoot(child) != findRoot(parent));
        access(child); access(parent);
        assert(!child->child[0] && !child->child[0]);

        child->attach(parent, 0);
}
```

For now, we can just $access$ u and v. We get a splay tree with only u (since it is a root) and another one containing v and its ancestors. We can just set v to be the left chid of u. This means that v has smaller depth than u. In other words, u becomes the child of v in the represented tree.

$Cut(u)$ removes the edge connecting u with its parent. It assumes u has a parent.

We simply access u, making the path from the root to u a preferred path. In the corresponding splay tree, we have u at the root and its left subtree represent its ancestors. We can just detach u from its left subtree. We need to set u as the path parent of the left subtree since it is now a splay tree on its own.

```
void cut(int id) {
        Node* u = nodes[id];
        access(u);
        assert(u->child[0]);
        u->child[0]->splayTreeParent = nullptr;
        u->child[0] = nullptr;
}
```

We also implemented a version of cut that takes two vertices and removes the edge between them. We just have to find which vertex is the child and call the above cut on that vertex.

$FindRoot(u)$ finds the root of the represented tree that contains u.

We access u, making the path from the root to u a preferred path. So, the splay tree that represents that path will have the root as the leftmost node since the root has depth 0. We just find that leftmost node and return it.

```
int findRoot(int id) {
        Node* u = nodes[id];
        access(u);
        Node* res = u->findMin();
        access(res);
        return res->id;
}
```

Since $access(u)$ takes $O(\log n)$ amortized time, every other operation has the same time complexity.

V. Rerooting

To be able to use $Link(u, v)$ for any u, v we will implement a rerooting function that allows us to set any node u to be the root of its tree.

The idea is to augment our splay tree node with a Boolean $reverse$. It is set to true if the 2 subtrees of the node should be swapped. In other words, the parent of the node should be its child, and its child should be its parent.

When we want to set a node u to be the root (especially the node u when calling $Link(u, v)$), we can first access it. We now have it as a root of a splay

tree with no right subtree. Its left subtree are its ancestors; what we want is to make those its descendent.

We will detach the left subtree of u and flip the reverse Boolean of the left child of u. This means that all nodes in this subtree can now be assumed to have bigger depth than u in the represented tree. Note that we cannot just flip the children of u since we also have to flip the children of the children of u and so on.

```
void link(int parentId, int childId) {
        Node* parent = nodes[parentId], * child = nodes[childId];
        assert(findRoot(child->id) != findRoot(parent->id));
        access(child); access(parent);

        Node* lChild = child->child[0];
        if (lChild) {
                lChild->reverse = !lChild->reverse;
                child->detachChild(0);
        }

        child->attach(parent, 0);
}
```

For now, we just changed a Boolean. When accessing a node, we can check if that Boolean at a node is true, in which case we do a DFS on its subtree. If a node in the DFS has $reverse = true$, we swap both its children and then we flip the $reverse$ value for both of those and we recur. Swapping children and propagating the Boolean are implemented in this function.

```
void applyReverseLazy() {
        if (!reverse) { return; }
        reverse = false;
        swap(child[0], child[1]);
        if (child[0]) { child[0]->reverse = !child[0]->reverse; }
        if (child[1]) { child[1]->reverse = !child[1]->reverse; }
}
```

The problem here is that this could take $O(n)$ time.

To fix this, we use a technique called **lazy propagation**. We only swap children and propagate the value when we are accessing the current node. In other words, we only call the $applyReverseLazy()$ function when we are

using the current node in some other function. The key idea here is that we are doing constant extra work in the previous functions. Hence, everything remains $O(\log n)$. This is still correct since the nodes we are using have the correct children.

Since we always $access$ a node before using it, and $access$ always splays a node, we can just modify $splay$ to call the above function. We add to it the following code that iterates over all the ancestors of the current node, calling $applyReverseLazy()$ on them:

```
stack<Node*> ancestors;
Node* cur = this;
while (cur) { ancestors.push(cur); cur = cur->splayTreeParent; }
while (ancestors.size()) {
        ancestors.top()->applyReverseLazy(); ancestors.pop();
}
```

While this might seem linear at first sight, remember that this code is used in a splay tree where the height is logarithmic (on average). Moreover, in the splay function we are iterating over the parents in all cases. So, we just doubled the work done: the time complexity does not change.

VI. Path Queries

To be able to handle any path queries, we first implemented the path aggregate function:

$PathAggregate(u)$ finds some value for the path from the root to u.

```
NodeVal* pathAggregate(int id) {
        Node* u = nodes[id];
        access(u);
        return u->nodeVal;
}
```

It just calls $access$ on u and then returns the value at u. The idea is that $access(u)$ makes the path from the root to u preferred, thus the corresponding splay tree has the nodes of the unique path from the root to u. Our splay tree functions maintain the subtree values when rotating nodes in the splay tree. Thus, at the root of the splay tree, we have a correct subtree value. We can modify the $update$ function and the attributes of $NodeVal$ to have any subtree value we want. A simple example is to return the subtree size which would represent the distance from the root to u.

Now, to be able to handle more interesting queries such as finding the distance between any two nodes, we will need to find the Lowest Common Ancestor of u and v. In that case, the distance between u and v becomes $dist(u,v) = dist(u,LCA) + dist(v,LCA) = dist(u,root) + dist(v,root) - 2 \times dist(LCA,root)$. We reduced our problem to finding the LCA and the distances from the root.

To find the LCA, we first $access(u)$. Then, if we were to call $access(v)$, we can notice that the last used path parent pointer will be the LCA. This is because it represents the deepest common node between the path from the root to u and the path from the root to v. Since we already implemented $access(v)$ to return the last seen path parent pointer, we can just return its return value.

```
int LCA(int id1, int id2) {
        Node* u = nodes[id1], * v = nodes[id2];
        if (findRoot(u->id) != findRoot(v->id)) { return -1; }
        access(u);
        return access(v)->id;
}
```

Both functions only call $access(\,)$. Hence, they also run in $O(\log n)$ amortized time.

VII. Testing and Problem Solving

We first ran some local tests that can be checked by hand to make sure that the functions behave correctly on small test cases.

Then, we checked our connectivity functions on SPOJ and Codeforces. We also tested our LCA functions on SPOJ.

Finally, we modified $NodeVal$ to keep track of the min and max values in the subtree and tested our path queries on Timus.

V. Analysis

After we successfully implemented both the Euler tour tree and the Link Cut Tree we thought about comparing their speed performance in the common functions which are: link, cut and isConnected. To do so we generated random graphs and random queries and then tested both implementations on them. Such tests allowed us not only to compare them both but to also check their running time experimentally since proving their complexity is not an easy job to do.

1. Testing Data Generation

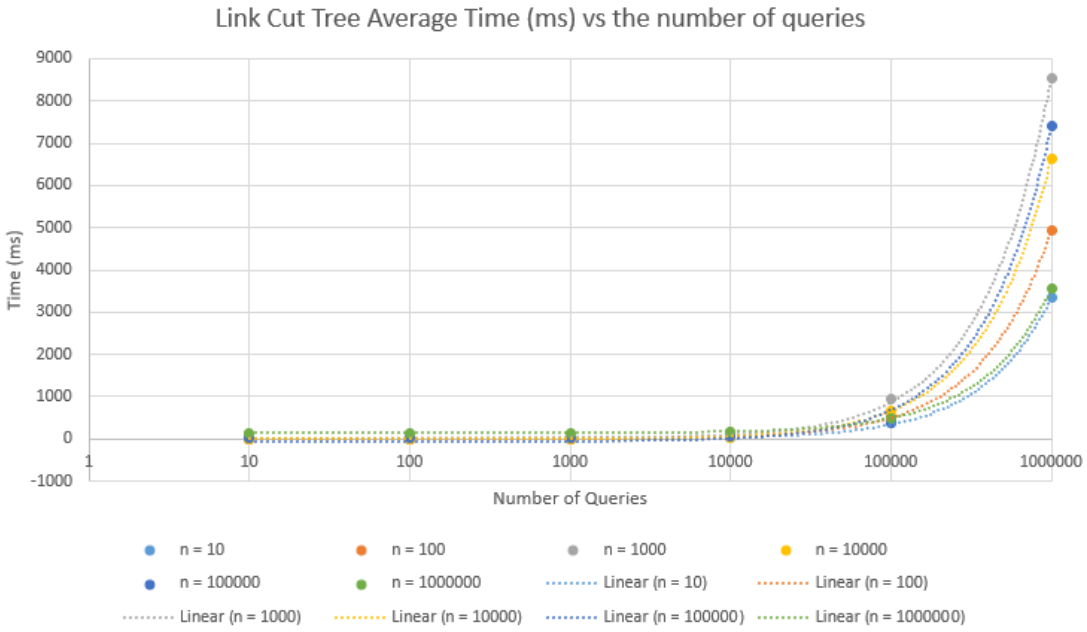
We generated 36 files which will be used by both implementations. We varied the size of the graph and the number of queries from 10 to 10^6 by factors of 10. Thus each file corresponds to a size of tree “n” and a number of queries “q”, in addition each file contains 10 test cases, i.e. we will repeat the experiment for same combination of n and q 10 times with different values for each test case.

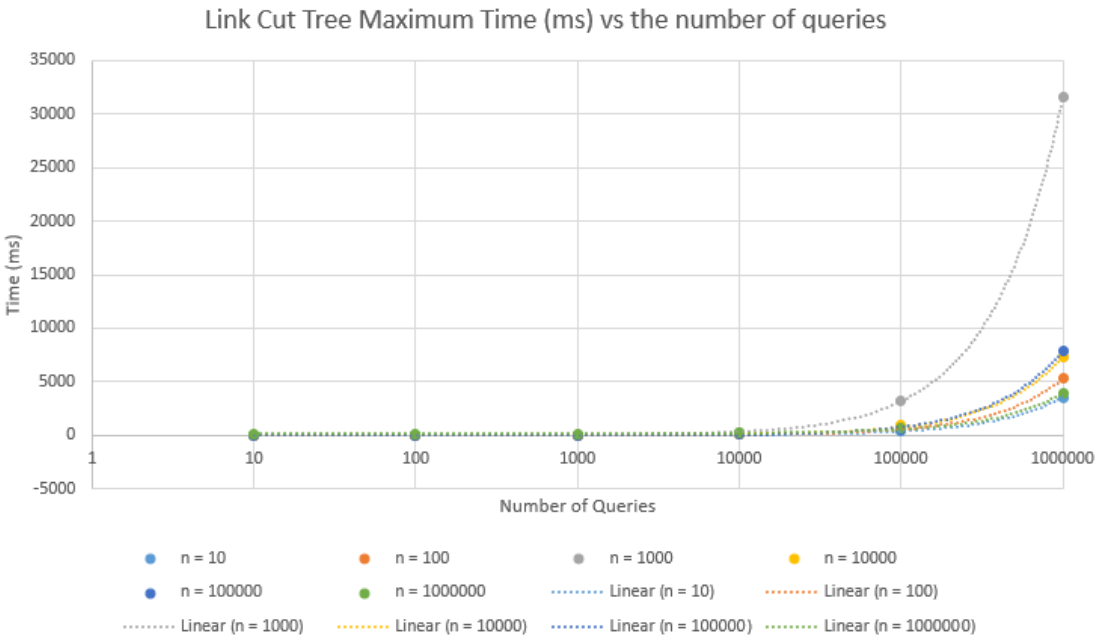
```
static void GenerateTreeTestCase(int  sizeOfTree, int numberOfQueries, ofstream & fout) {
	fout << sizeOfTree << " " << numberOfQueries << "\n";
	LinkCutTree lct;
	for (int i = 0; i < sizeOfTree; i++) lct.addNode(new NodeVal());
	vector<pair<int, int>> edges;
	mt19937 rng(chrono::steady_clock::now().time_since_epoch().count());
	for (int q = 0; q < numberOfQueries; q++) {
		int decision = rng() % 4;
		if (decision < 2) {
			// add edge
			if (edges.size() == sizeOfTree - 1) {
				q--;
				continue;
			}
			fout << "1 "<<GetTwoNodesNotConnected(sizeOfTree, lct, edges, rng)<<"\n";
		}
		else if (decision == 2) {
			// remove edges
			if (edges.size() == 0) {
				q--;
				continue;
			}
			fout << "2 " << GetTwoNodesConnected(sizeOfTree, lct, edges, rng) << "\n";
		}
		else {
			// query
			fout << "3 " << rng() % sizeOfTree << " "<< rng() % sizeOfTree << "\n";
		}
	}
	for (int i = 0; i < sizeOfTree; i++) delete lct.nodes[i];
}
```

Above is the c++ code we used to generate trees, as you can see we increased the probability of adding an edge over removing an edge in order to guarantee that on average we have more edges being added than edges being removed.

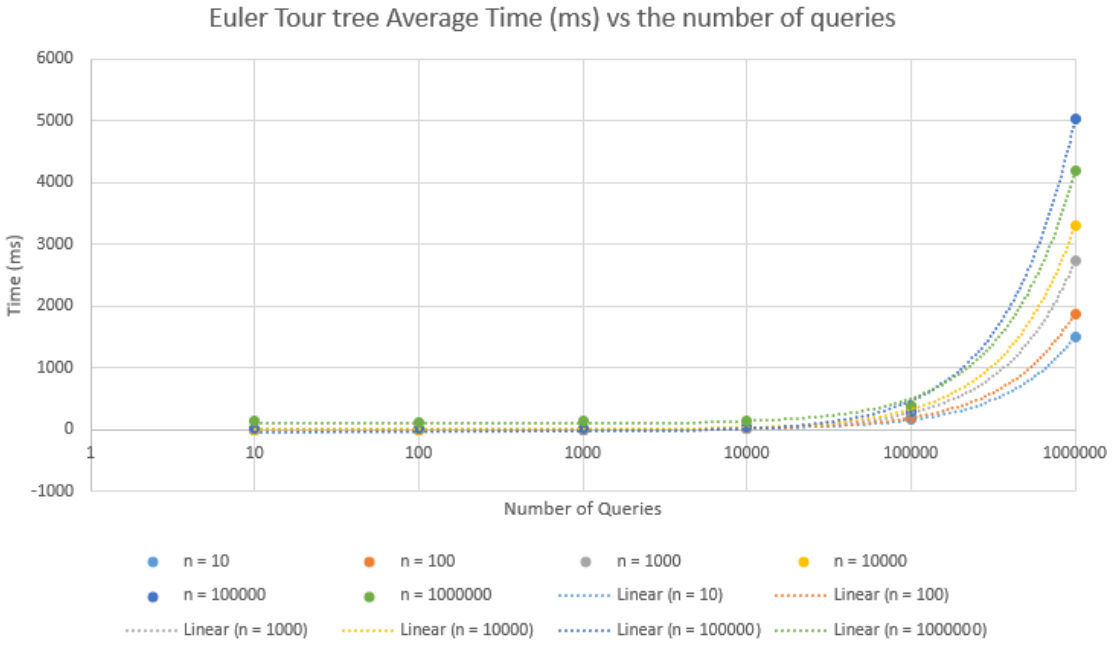
2. Results

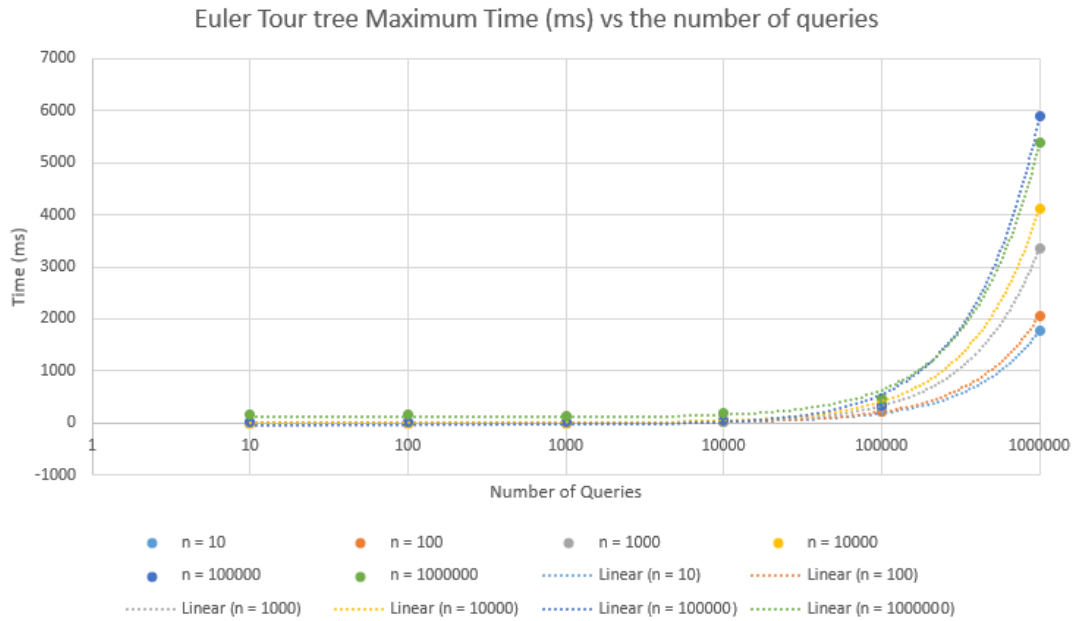
The detailed tables of the experiment can be found in the appendix, we will show here some interesting plots instead.





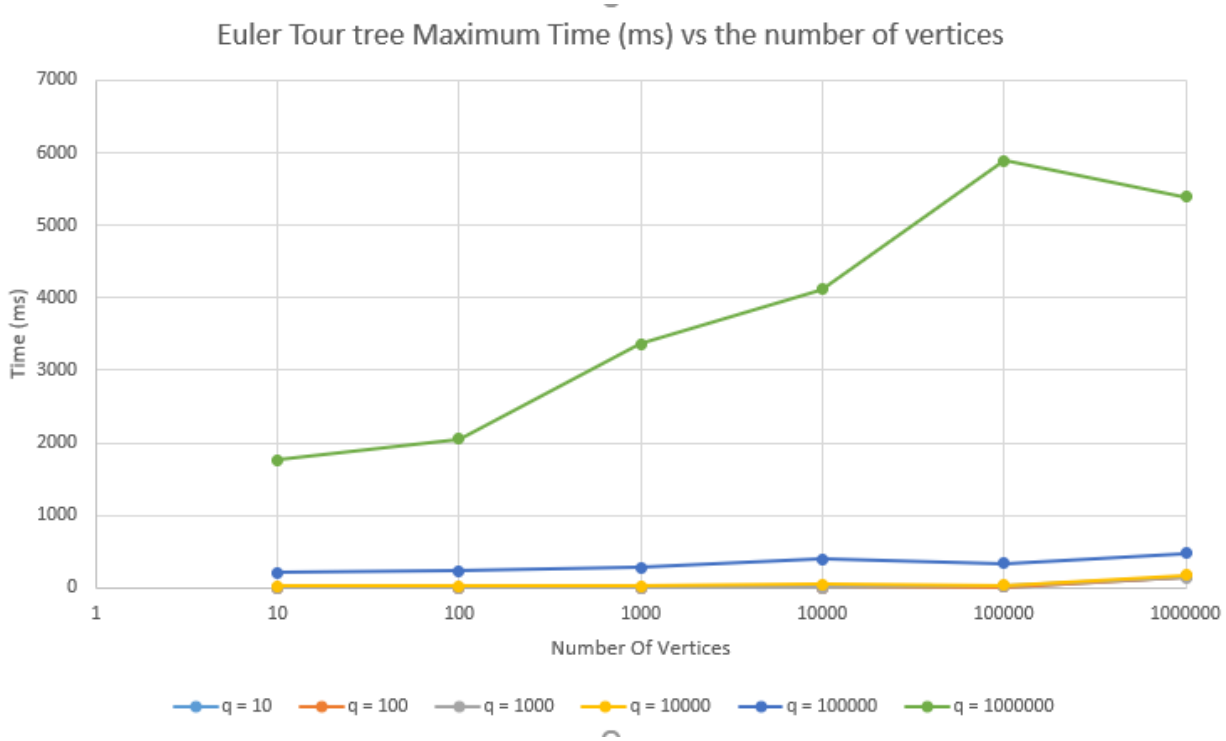
From the above plot we can see that the average and the maximum running time of the link cut tree is exponential with respect to the logarithm of the number of queries and thus it is close to linear with respect to the number of queries. This confirms the O(N + q log n) theoretical running time.

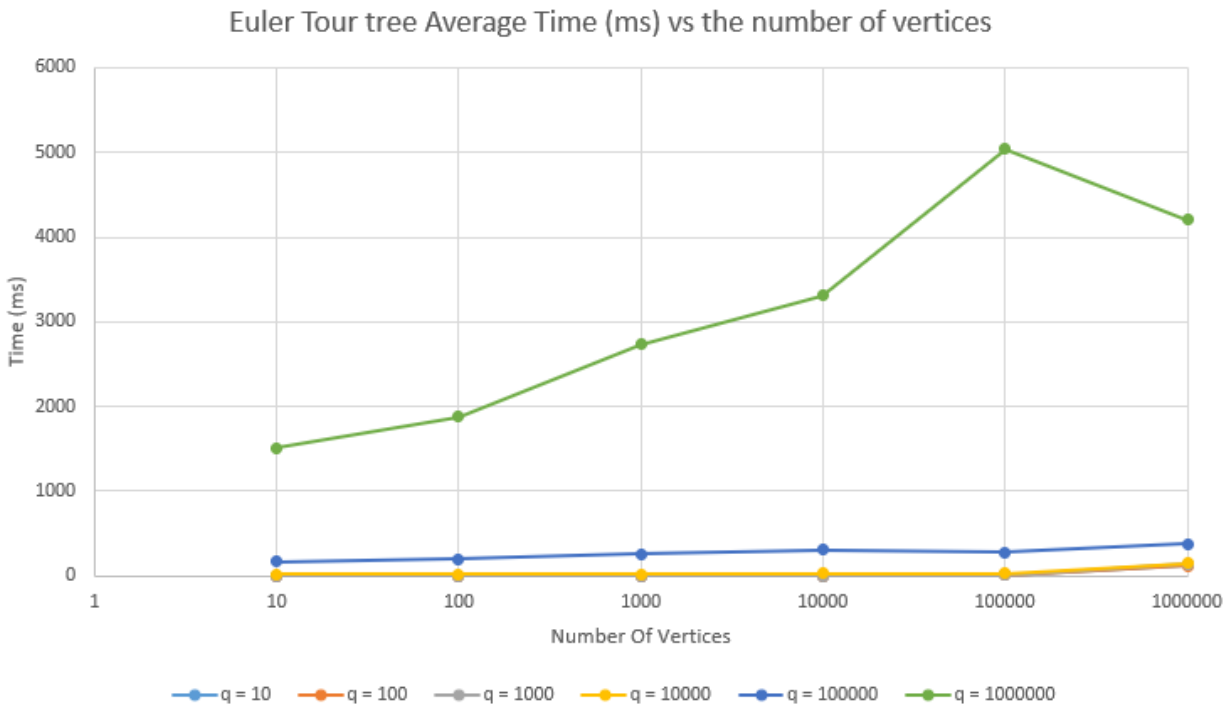


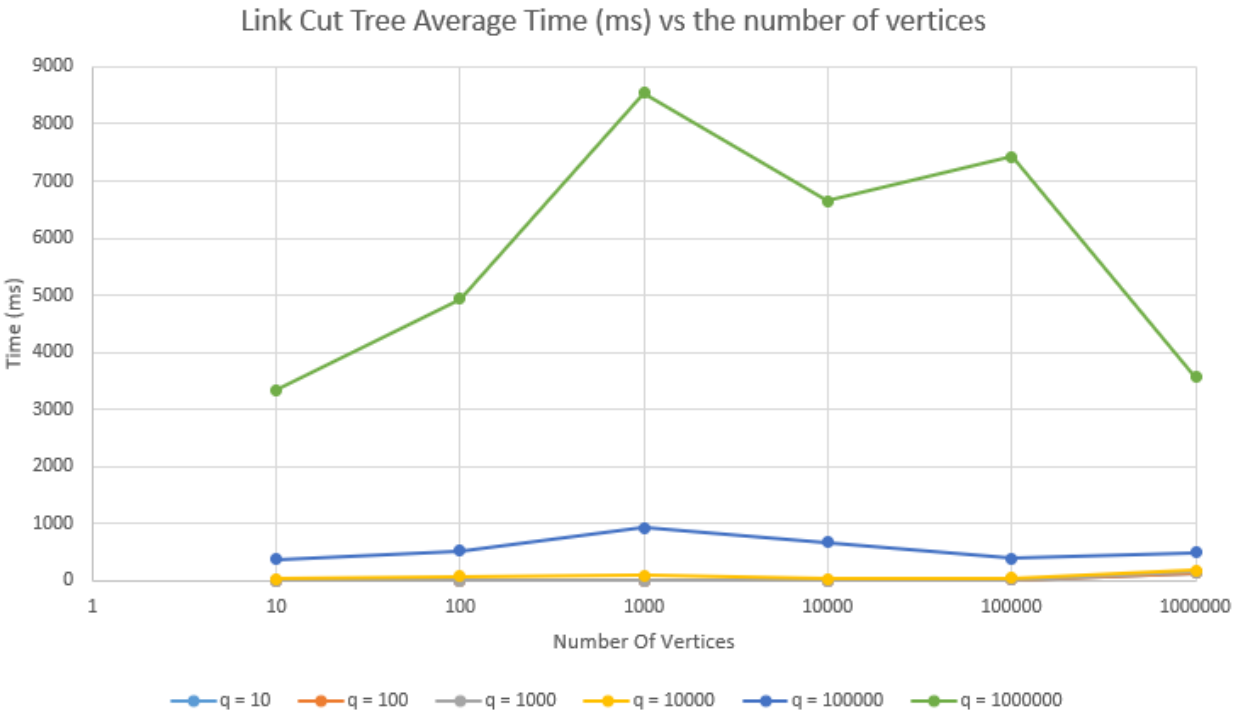


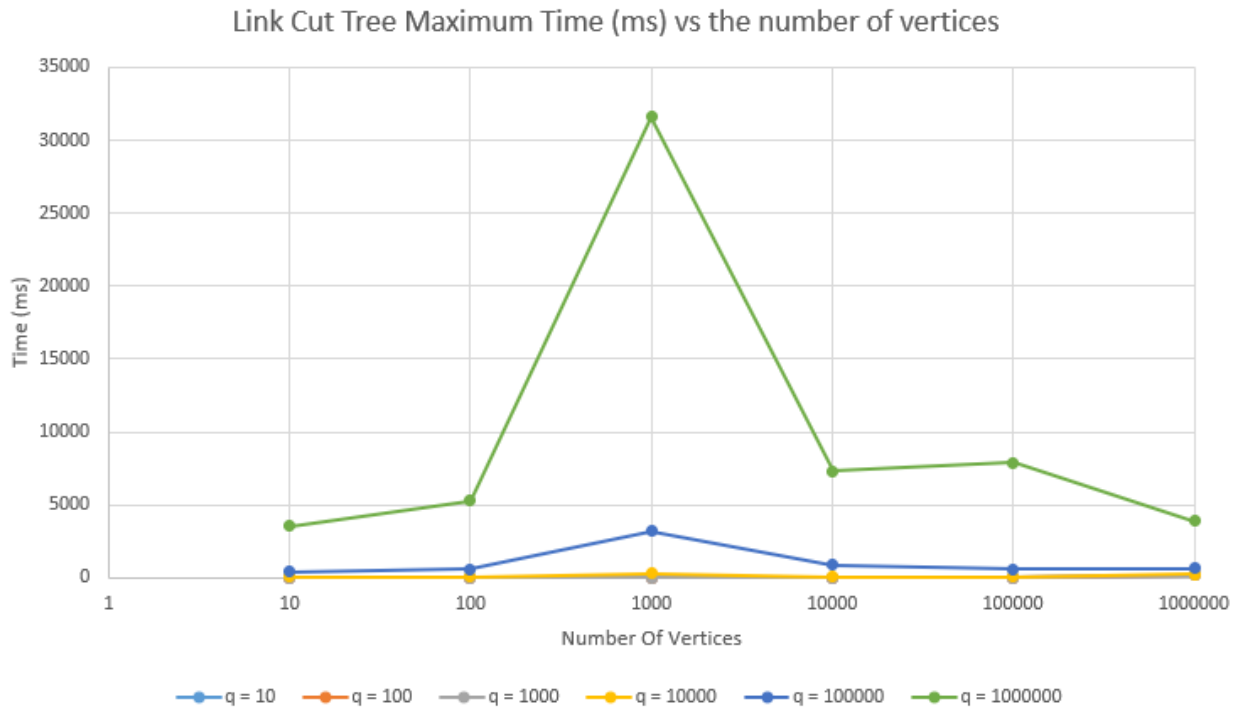
From the above plots we can see that the Euler tour tree have the same behavior as the link cut tree but with a better constant factor both on average and worst case.

What about if we plotted the performance with respect to the size of the graph instead of the number of queries?









We can see from the above plots that the running time is dominated by the number of queries and not the number of vertices. If we fix the number of queries we

notice that the Average and maximum times are roughly constant when q <= 1e6 no matter how we vary the number of vertices (although we multiplied n by 1e6 we don’t see any noticeable changes in the time).

Thus, we first confirm the theoretical running times observed with experimental evidence, and second, we suggest that the Euler Tour tree has a better constant factor however the Euler tour tree does not support queries on Path like the Link Cut Tree.

VI. Third Method: Divide & Conquer + Persistent DSU

After we finished solving the problem for trees, we tried solving it for general graphs. Due to shortage in time, we added an additional restriction which is that the queries will be offline: we will be given all queries first and then answer them all at the end. The algorithm we used is a divide and conquer algorithm along with a persistent disjoint set union data structure and it was mainly inspired by a blog on Codeforces.com [3].

1. Prerequisite: Persistent Disjoint Set Union Data Structure (DSU)

DSU is a data structure which allows us to join any 2 sets and answer in logarithmic time whether 2 elements belong to the same set or not.

Here we can see a simple implementation of union and get, union simply joins 2 sets and get returns the id of the set containing an element.

```
union(a, b):
  a = get(a)
  b = get(b)
  if size[a] > size[b]:
    swap(a, b)
  p[a] = b
  size[b] += size[a]
```

```
get(a):
  while a != p[a]:
    a = p[a]
  return a
```

A persistent DSU is a DSU equipped with the possibility of rolling back changes and thus going back to a previous state in constant time amortized. The idea is that we are only applying 2 changes in union(a, b) which are changing size[b] and p[a]. so we can simply store the old values before updating them and thus revert a change when needed.

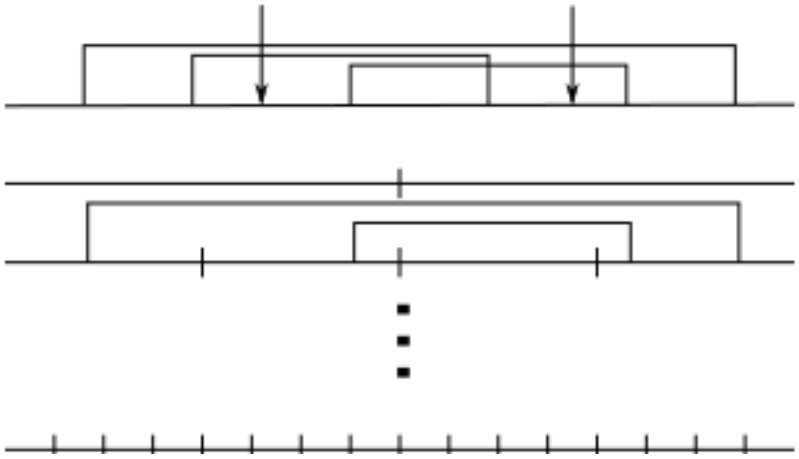
2. Preprocessing

First we suppose that each added edge is removed, we can ensure this by removing all remaining edges at the end.

Once we do this we replace each pair of link(u, v, t1), cut(u, v, t2) with another query we will call edge(u, v, t1, t2) which simply means that the edge between u and v is present between t1 and t2. Keep in mind that link(u, v, t1) means that we linked u and v at time t1 and all this information is provided to us since the queries are offline, i.e. we have all the queries at the beginning.

3. Divide & Conquer Algorithm

Now we start dividing our timelines into halves, as we go down in our recursion we will maintain a persistent DSU with us which contains all edges that cover fully the segment we are in. In other words, if we are working on the segment [2, 5] our DSU will contain all edges that starts before 2 and ends after 5. This will ensure that as we go down, all edges added already can be used by all subsegments. Now when we reach a segment of length 1, i.e. we are at [t, t], we use our DSU and answer all isConnected(u, v, t).



Now once we finish a level and we want to go back from our recursive call in a subsegment we rollback the changes we did in this subsegment and then the original call will have his DSU untouched and ready to use for the other subsegment.

4. Complexity and Testing:

If we assume that we have m queries, the complexity is O(m * log m * log n). The reason is that each DSU query takes O(log n), rolling back is constant amortized, and we have log m levels so in total we get O(m * log m * log n).

We tested this algorithm on both SPOJ and Codeforces.

VII. Appendix

Average Time (ms) Link-Cut Tree						
	n=10	n=100	n=1000	n=1e4	n=1e5	n=1e6
q=10	0	0	0.2	1.3	13.5	140.1
q=100	0.3	0.3	0.8	1.6	15.7	142.4
q=1000	3.9	4.6	4.6	4.1	18.6	145.5
q=1e4	37.7	69.6	91.2	36.2	50.5	176.4
q=1e5	367.1	520.1	930	666.9	385.4	495.5
q=1e6	3335.5	4942.7	8551.2	6648.3	7421.4	3559

Max Time (ms) Link-Cut Tree						
	n=10	n=100	n=1000	n=1e4	n=1e5	n=1e6
q=10	0	0	1	2	16	155
q=100	1	1	2	2	22	156
q=1000	6	6	14	5	23	162
q=1e4	44	84	303	52	64	211
q=1e5	440	608	3178	878	570	627
q=1e6	3549	5255	31588	7304	7890	3908

Average Time (ms) ETT						
	n=10	n=100	n=1000	n=1e4	n=1e5	n=1e6
q=10	0	0.1	0.2	1.4	11.7	129.3
q=100	0.1	0.3	0.3	1.7	11.8	120
q=1000	1.6	1.9	2	3.1	13.4	136.7
q=1e4	20	21.4	24.5	30.1	32.5	150.8
q=1e5	174.2	199.3	259.5	308.3	277.8	378.5
q=1e6	1508	1881.8	2729.6	3310.3	5043.8	4204.8

Max Time (ms) ETT						
	n=10	n=100	n=1000	n=1e4	n=1e5	n=1e6
q=10	0	1	1	2	15	150
q=100	1	1	1	3	17	153
q=1000	2	2	3	4	19	145
q=1e4	26	26	29	48	35	178
q=1e5	215	229	282	398	336	474
q=1e6	1762	2055	3365	4122	5893	5392

VIII. References

[1] Daniel D. Sleator and Robert Endre Tarjan. 1983. *A data structure for dynamic trees*. J. Comput. Syst. Sci. 26, 3 (June 1983), 362–391. DOI: https://doi.org/10.1016/0022-0000(83)90006-5

[2] Daniel Dominic Sleator and Robert Endre Tarjan. 1985. *Self-adjusting binary search trees*. J. ACM 32, 3 (July 1985), 652–686. DOI: https://doi.org/10.1145/3828.3835

[3] https://codeforces.com/edu/course/2/lesson/7/3

[4] https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-851-advanced-data-structures-spring-2012/calendar-and-notes/MIT6_851S12_L20.pdf

[5] https://codeforces.com/blog/entry/53265