CS302 PROJECT

DYNAMIC CONNECTIVITY

TEAM MEMBERS :

Group-6

Divya Gola - 2022CSB1080
Himanshi Wanjari - 2022CSB1085
Gachula Varsha - 2022CSB1082
Palla Santhoshi Shalini-2022CSB1100

PROBLEM STATEMENT

Our approach leverages dynamic graphs, which allow the representation of changes in underlying information over time, thereby extending the capabilities of traditional graphs. We introduce a fully dynamic algorithm designed to efficiently maintain connectivity, specifically tailored for scenarios involving edge insertions or deletions. In the context of our algorithm, "fully dynamic" signifies the ability to update the graph solely through the insertion or deletion of edges, without impacting the vertices.

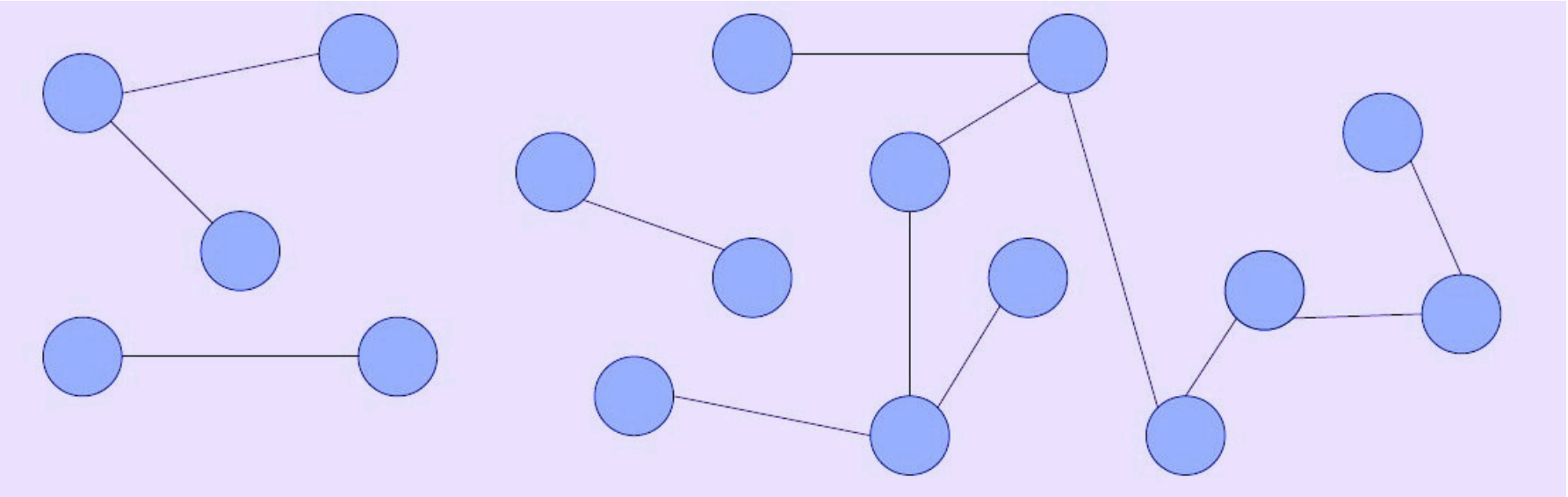
Let's denote the number of nodes in the graph as n. For a sequence of operations $\Omega(m0)$, where $m0$ is the number of edges in the initial graph, the expected time complexity for p updates is $O(p.\ log^2 n)$ for connectivity maintenance.

Dynamic Connectivity in Forests

Considering the following special-case of the dynamic connectivity problem:

Maintain an undirected *forest F* so that edges may be inserted an deleted and connectivity queries may be answered efficiently.

Each deleted edge splits a tree in two; each added edge joins two trees and never closes a cycle [the forest maintains the minimum spanning trees].



EULER TOUR

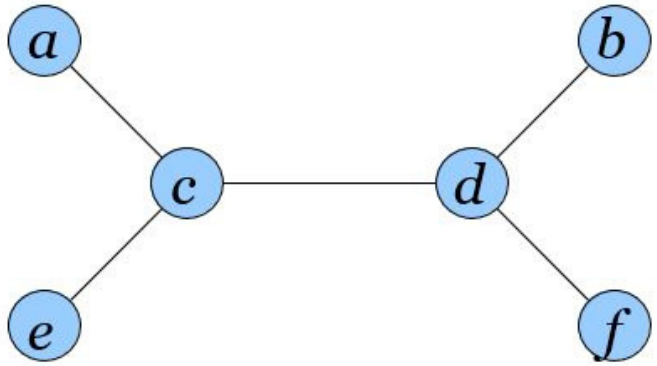
To solve the dynamic connectivity problem in forest Henzinger and King proposed the Euler Tour Tree Data Structure.
An Euler tour is a path through a graph G that visits every edge exactly once.
Euler Tour Tree Data Structure stores the Euler Tour of a graph and supports these three operations:

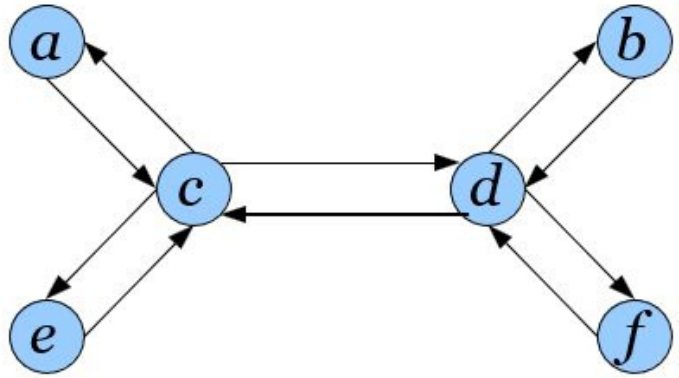
- **link(u, v)**: Add in edge uv. The assumption is that u and v are in separate trees.
- **cut(u, v):** Cut the edge uv. The assumption is that the edge exists in the forest.
- **are-connected(u, v):** Return whether u and v are connected.

The data structure we'll develop can perform these operations time O(log n) each.

Euler Tours on Trees

To make a tree Euler Traversable we replace each edge {u,v} of the tree with (u,v) and (v,u). Representing a tree this way we can ensure that the tree has an Euler Tour.



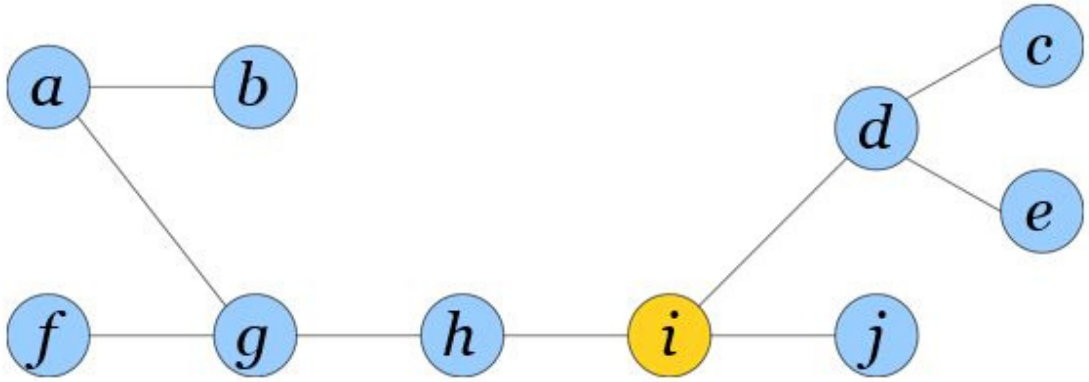


ac cd db bd df fd dc ce ec ca

Rerooting a Tour

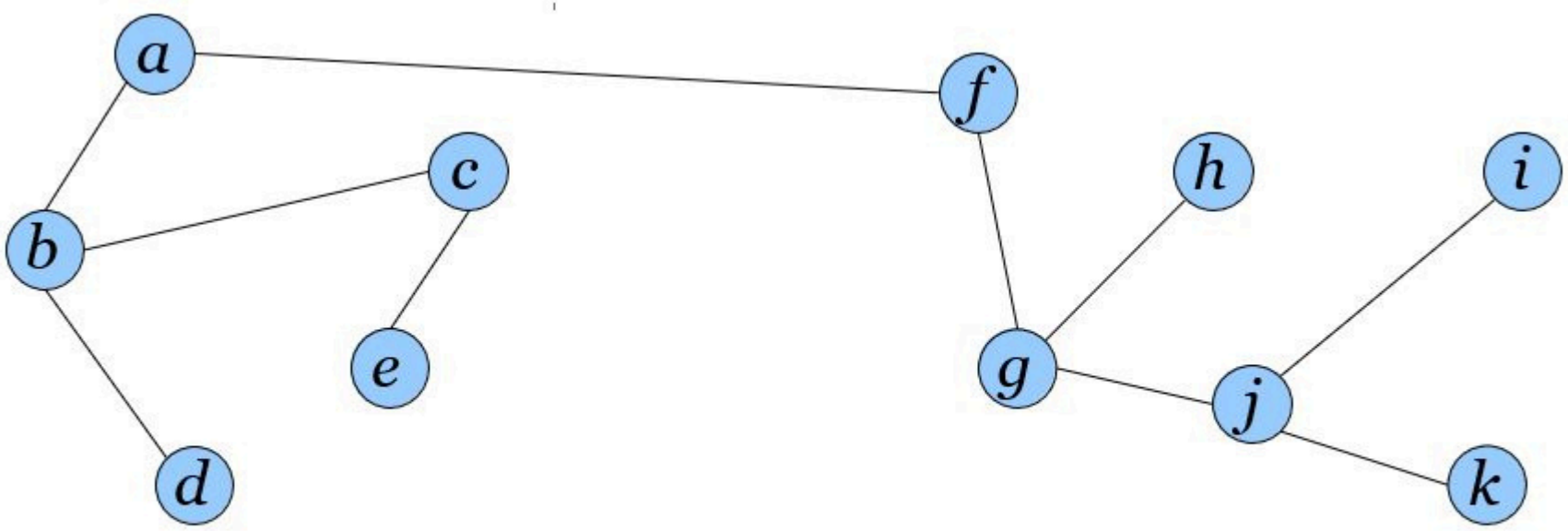
To perform *makeroot*(*x*):

- Pick any edge rx leaving our new start node r.
- Split the tour into A and B, where A consists of everything up to but not including rx and B consists of everything from rx forward.
- Concatenate B A.



Euler Tours and Dynamic Trees

Given two trees T_1 and T_2, where $u \in T_1$ and $v \in T_2$, executing $link(u, v)$ links the trees together by adding edge uv.



ab bd db bc ce ec cb ba af fg gj jk kj ji ij jg gh hg gf fa

Euler Tours and Dynamic Trees

Euler Tours and Dynamic Trees

Given two trees T_1 and T_2, where $u \in T_1$ and $v \in T_2$, executing link(u, v) links the trees together by adding edge uv.

To link(u, v):

Let E_1 and E_2 be Euler tours of T_1 and T_2, respectively.

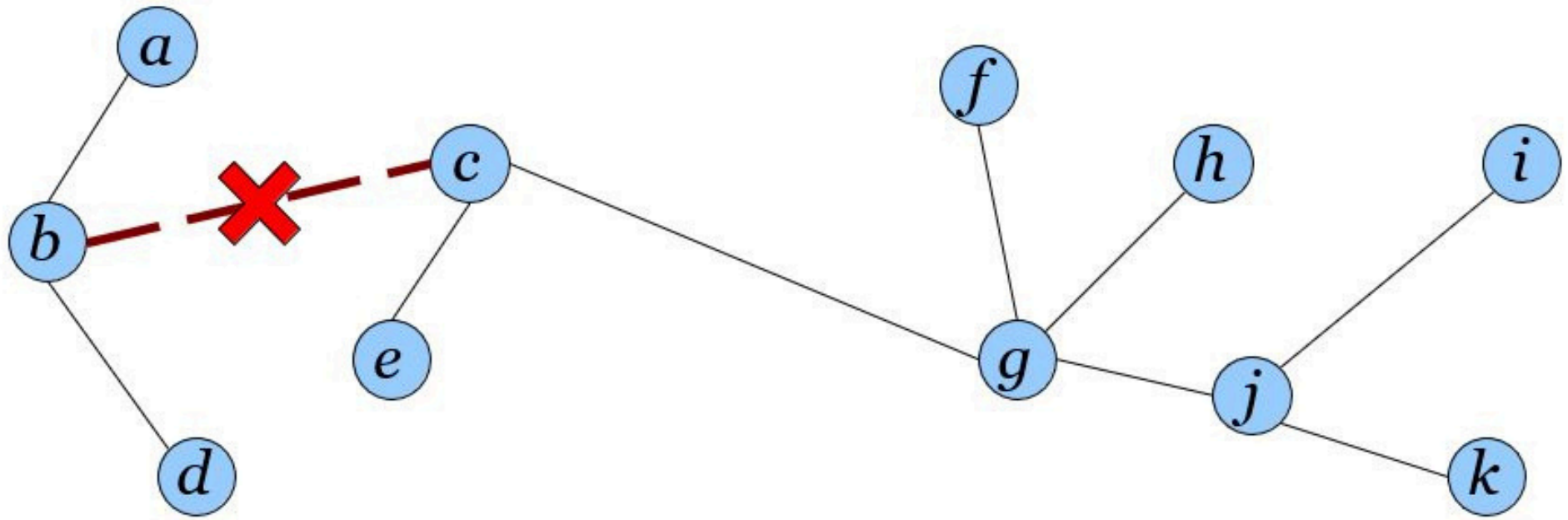
makeroot(u)

makeroot(v)

Concatenate E_1 uv E_2 vu.

Euler Tours [Cut Operation]

Given a tree *T*, executing *cut*(*u*, *v*) cuts the edge *uv* from the tree (assuming it exists).



ce ec cb ba ab bd db bc cg gh hg gf fg gj jk kj ji ij jg gc

Euler Tour [Cut operation]

Given a tree T, executing cut(u, v) cuts the edge uv from the tree (assuming it exists).

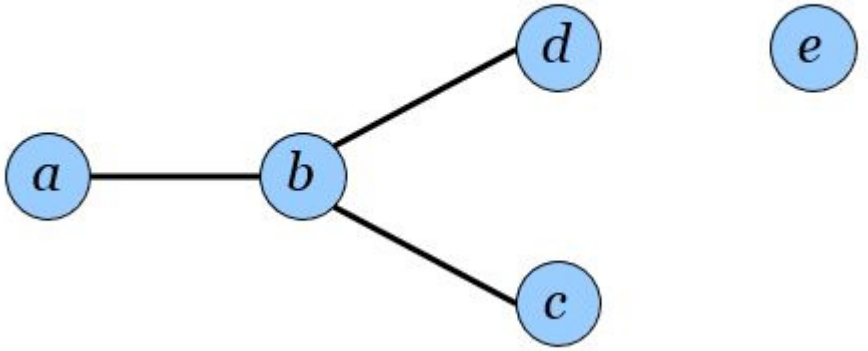
To perform cut(u, v):

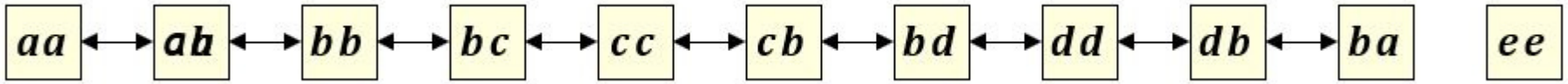
- Let E be the Euler tour containing uv and vu.
- Remove uv and vu from E to form E_1, E_2, and E_3.
- Then E_1E_3 and E_2 are Euler tours of the two new trees.

Implementing Euler Tour Trees

Idea 1: Use doubly-linked lists, plus an auxiliary hash table / BST to locate edges.

Assuming we have a hash table telling us where edges are, we can split, join, and rotate tours in time O(1). But finding if two nodes are connected takes O(n).





Implementing Euler Tour Trees

In Euler Tour Tree data structure we represent the Euler Tour splay trees! They support these operations in amortized time O(log *n*).

The above operations can be written as a series of constant number of split() and merge() operations on the binary trees. This ensures that these operations take *O*(log(n)) time to execute.

- makeRoot(u) := 1 split() and 1 merge()
- link(u,v) := 2 makeRoot(u) and 3 merge()
- cut(u,v) := 3 split() and 1 merge()

INVARIANTS

Invariant 1 : Every connected component of G_i has at most 2 vertices.

Invariant 2 : $0 \subseteq F1 \subseteq F2 \subseteq \ldots \subseteq F_{\log n}$.In other words $F_i = F_{\log n} \cap G_i$ and $F_{\log n}$ is the minimum spanning forest of $G_{\log n}$,where the weight of an edge is it's level.

insertEdge(u,v)

insert (u, v) into E;

l((u, v)) ← log(n); // set level of (u, v) as log(n)

if u and v not connected in Flog(n) then

    Flogn.insert(u, v);

    insert (u, v) in $\text{TreeEdge}^{\log(n)}$;

else

    insert (u, v) in NonTreeEdgelog(n);

removeEdge(u,v)

remove (u,v) from E

level = $l((u, v))$
If (u,v) is in NonTreeEdge $_{level}$

then remove (u,v) from NonTreeEdge $_{level}$

else

foreach i from level to logn, as long as no replacement found do

Fi.delete(u, v);

Tu ← tree of Fi that u is in;
Tv ← tree of Fi that v is in;

if |Tu| > |Tv| then
swap u and v;

foreach (x, y) ∈ F i.listT (u) do
//Go through tree edges of level i in Tu;
decrease-level(x, y);

foreach (x, y) ∈ Fi.listNT (u) **do**
u;
//Go through non-tree edges of level i in T

if y ∉ T_V then
decrease-level(x, y);

else if no replacement found yet **then**
found replacement;
foreach j = i to logn **do**
Fj .delete(u, v);
F j .insert(x, y);
STOP.

decrease-level((u, v))

```
l ← level(u, v);
level(u, v) ← l – 1;

Fl.remove(u, v);
if (u, v) ∈ Fl then
    Fl-1.insert(u, v);
    remove (u, v) from TreeEdgel;
    insert (u, v) into TreeEdgel-1;
else
    remove (u, v) from NonTreeEdgel;
    insert (u, v) into NonTreeEdgel-1;
```

isConnected(u,v)

if $F_{log(n)}$.root(u) = $F_{log(n)}$.root(v)

    return true;

else

    return false;

Proof

Invariant 1 **: Every connected component of G_i has at most 2^i vertices.**

To Prove **: Size of any tree of F_i <= 2^i for all 1 <= i <= log(n).**

Proof : For i = log(n), $F_{log(n)}$ has 1 connected component i.e. the whole graph of size n. Size($F_{log(n)}$) <= $2^{log(n)}$ = n . *TRUE*

Edges in F_i are formed when a subtree of some tree in F_{i+1} is pushed down due to a delete operation.

After deletion let T_v and T_w be the subtrees of some tree T in F_{i+1} formed after deletion of a tree edge in tree T. Without loss of generality let T_v be the smaller subtree in which all level i+1 tree edges are pushed down to level i as part of delete operation.

Proof

Let T’ be the new connected component formed by adding these pushed down edges to F_i

By *Invariant 2*:

F_i is a sub-forest of F_{i+1}

=> T’ is a subtree of T_v with all i+1 level tree edges pushed down to level i.

=> T’ is actually T_v

Proof

We will further proceed by induction.

Let size(F_i) denote the maximum size of any tree in F_i.

Inductive Step: Before deletion assume size (F_{i+1}) <= 2^{i+1} and size(F_i) <= 2^i.

Then after deletion size(F_{i+1}) <= 2^{i+1} as size of largest component may remains same or decreases after edge deletion.

Since new edges get added to T' in F_i, size(F_i) might change.

$$Size(F_i) <= max(2^i, |T'|)$$

Proof

But $|T| <= 2_{i+1}$ as T was a tree in F_{i+1} and size(F_{i+1}) <= 2^{i+1}

$$|T_v| + |T_w| <= 2^{i+1}$$

$$|T_v| <= 2^i \text{ as } |T_v| <= |T_w|$$

=> $|T'| <= 2^i$ (as $|T'| = |T_v|$)

$$\text{Size}(F_i) <= \max(2_i , |T'|)$$

$$\text{Size}(F_i) <= \max(2^i, 2^i) = 2^i$$

Proof

Invariant 2 : $F_0 \subseteq F_1 \subseteq F_2 \subseteq \ldots\ldots F_{\log(n)}$. In other words $F_i = F_{\log(n)} \cap G_i$ and $F_{\log(n)}$ is the minimum spanning forest of $G_{\log(n)}$, where the weight of an edge is its level.

Proof: Hierarchical decomposition of edges in levels happens as a side effect of deletion of a tree edge. This is because level of edges gets decreased only when some edge gets deleted.
Formally an edge r of level i gets pushed down to level i - 1 when some tree edge e of level l(l <= i) is deleted and r is a tree/non-tree edge with both its end points in the smaller subtree formed after deletion of e.

Proof

Invariant 2 is not violated by the two operations insertEdge(u,v) and removeEdge(u,v)

a) *insertEdge(u,v):* Assume invariant 2 to be true before inserting (u,v).

Case 1: (u,v) is a non tree edge i.e. (u,v) is not added in $F_{log(n)}$. Invariant 2 holds true.

Case 2: (u,v) is a tree edge then (u,v) is added to $F_{log(n)}$ at level log(n). Since

additional edge is added in Flog(n) and other spanning forest remain same => $F_0 \subseteq F_1 \subseteq F_2 \subseteq \ldots F_{log(n)}$ remains true.

Proof

b) removeEdge(u,v): Assume that before deletion of edge (u,v) invariant 2 holds true i.e. $F_0 \subseteq F_1 \subseteq F_2 \subseteq ...F_{log(n)}$

Let level of (u,v) be *l* , the algorithm searches for a replacement edge in all levels i from *l* to log(n) until it finds one.

Iteration 1 (i = *l*) : All tree edges of level *l* are pushed down to level *l* - 1, adding new edges to F_{l-1} . Since these edges already exist in F_l => F_{l-1} remains $\subseteq F_l$. *Invariant 2 holds TRUE*

Proof

Iteration i : All tree edges of level i are pushed down to level i - 1, adding new edges to F_{i-1}.
Since these edges already exist in F_i => F_{i-1} remains $\subseteq F_i$. *Invariant 2 holds TRUE .*

Suppose replacement edge gets found at level i then this edge is added in F_i , F_{i+1} , ... $F_{log(n)}$
Invariant 2 holds TRUE .

=> $F_l \subseteq F_{l+1} \subseteq$ ..... $F_{log(n)}$ even after multiple push down of tree edges and insertion on replacement edge.

=> Invariant 2 holds TRUE after removeEdge(u,v).

Proof

To Prove: maximum level of an edge = log(n) (floor(log(n) to be specific)
Proof:
From invariant 1 we can say that at any point of time, at some level i,

Maximum size of any tree in F_i <= 2^i

=> At level i=0 maximum size of any tree in F_0 <= 1

=> Forest at level 0 consist of all single nodes
=> No edge at level 0 => An edge can have maximum level log(n).

Proof of Correctness

To Prove: Upon deletion of some edge (u,v) which was a tree edge that first appeared in level l, if there exists a replacement edge (x,y) , it's level has to $\in$ [l , log(n)]. If the replacement edge (x,y) exists, it will be found by the algorithm. Proof: Being a replacement edge means that there is a tree path P in $F_{log(n)}$ between x and y that includes (u,v) and (x,y) is a non tree edge.

There can be several cases :

Case 1: Inserting (x, y) when the x − y path P including (u, v) is already there. In this case, the level of (x, y) is set to log(n) and the level of (u,v) is at most log(n) => *l* (x,y) >= *l* (u,v)

Replacement edge (x,y) will be found at level log(n) in the first iteration of removeEdge(u,v).

Proof of Correctness

Case 2: The deletion of some edge e’ causes the insertion of (u, v) to complete P, and (u, v) was a replacement edge for e’. In this case, before the update there was a tree-path including e’ between x and y so that (x,y) is also a replacement edge for e’. We can assume via an induction on the number of updates done by the algorithm (base case: no updates), that l(x, y) >= l(e’). When removeEdge(e’) was searching for a replacement, it chose (u, v) instead of (x, y) (and both are viable since their levels were >= l(e’)) because to maintain invariant 2 the algorithm prefers lower level edges and thus l(x, y) >= l(u, v).

Proof of Correctness

Additionally: Suppose that (x, y) is a replacement for (u, v) where i = l(x, y) >= l(u, v) and at some point l(x, y) decreased to i + 1 due to the deletion of some edge e. Now, in Fi, e split some tree into T_v and T_w, where T_V is the smaller one. Both x and y are in T_v since only replacement edges with both endpoints in the smaller tree have their levels decreased by delete. This means that there is a tree path within T_V between x and y and since (u,v) is a tree edge on such a path, (u,v) must also be in T_V. Thus, either l(u,v) >= i - 1 and so after the decrease of l(x, y) we still have l(u, v) <= l(x, y), or l(u, v) = i and hence delete also decreased l(u, v) to i - 1 and again l(u, v) <= l(x, y). Hence we proved if there exists a replacement edge (x,y) , it's level has to $\in$ [l , log(n)].

Proof of Correctness

To find replacement edge (x,y) the algorithm searches each level from l to log(n) i.e. in order of increasing edge levels. This ensures that a replacement edge if found at a lower level will be preferred, hence preserving invariant 2. Since the algorithm searches all the non tree edges of level i (l <= i <= log(n)) which have only one end point in the smaller subtree of the two subtrees formed after deletion of (u,v) in F_i, the replacement if exists, will be found. If the replacement edge does not exist algorithm updates all the spanning forest upto level log(n).

Running Time Analysis

isConnected(u,v): returns true if nodes u and v are connected in G and false otherwise.

It calls the findroot() function on ET Tree representation of $F_{\log(n)}$ and thus takes $O(\log(n))$ time.

Insert((u,v)) : inserts the edge connecting u and v into G. If u and v are already connected the edge is added to the edge map of G
Else the edge is added to the spanning forest at level logn as well

It takes O(log(n)) time to add the edge in ET Tree.

Running Time Analysis

Delete((u,v)): deletes edge connecting u and v and looks for a suitable replacement edge if needed.
If (u,v) is a non-tree edge it takes O(logn) time.
Else deletion from all the spanning forests where it is present

(F_i| level(u,v) <= i <= log(n)) takes $O(\log^2(n))$ time. And time required to push k total edges down a level is O(k*log(n)). Thus total time for deletion is $O(\log^2(n) + k*\log(n))$.

Running Time Analysis

Since absolute running time depends on k edges (which is a result of multiple insert and delete operations), we give an amortised running time of the whole program.

The proof utilises the Accounting Method of amortised analysis. Let us measure time in terms of tokens. We charge the insert operation extra tokens per edge which would later pay for the expensive delete operation.

Running Time Analysis

Lemma 1 : $\log^2 n$ tokens per edge are sufficient to pay for that edge being pushed down a level.

Proof:
Observation **:** Since only logn levels are there it implies that, at max each edge can undergo logn level decreases as part of delete operations on some edge.
Thus,
Cost of pushing down an edge a level below = c*log(n)
Cost of pushing down an edge <= c*log(n) * log(n) levels = $c*\log^2(n)$
This cost corresponds to $\log^2(n)$ tokens which will pay for level decreases in delete operations.

Running Time Analysis

Pushing down of edges is the operation being paid by tokens and hence we proceed with an invariant on this operation.
Actual cost of pushing down edges(c) = k*log(n) ; where k is the number of edges being pushed down in total
Amortised Cost (ĉ) = $m*\log^2(n)$; where m is number of inserted edges
Invariant : Since we are looking at per operation run time we will maintain the following cost invariant :

ĉ - c >= 0

Running Time Analysis

The cost invariant will be maintained because of the following :
Actual cost = k*log(n)
Since each edge can be decreased in level logn times (as there are logn levels) , we conclude

$$k <= m*\log(n)$$

This implies

$$k*\log(n) <= m*\log^2(n)$$

$$m*\log^2(n) - k*\log(n) >= 0$$

$$\hat{c} - c >= 0$$

This implies that even if all edges get reduced in level all the way, the invariant always holds.

Running Time Analysis

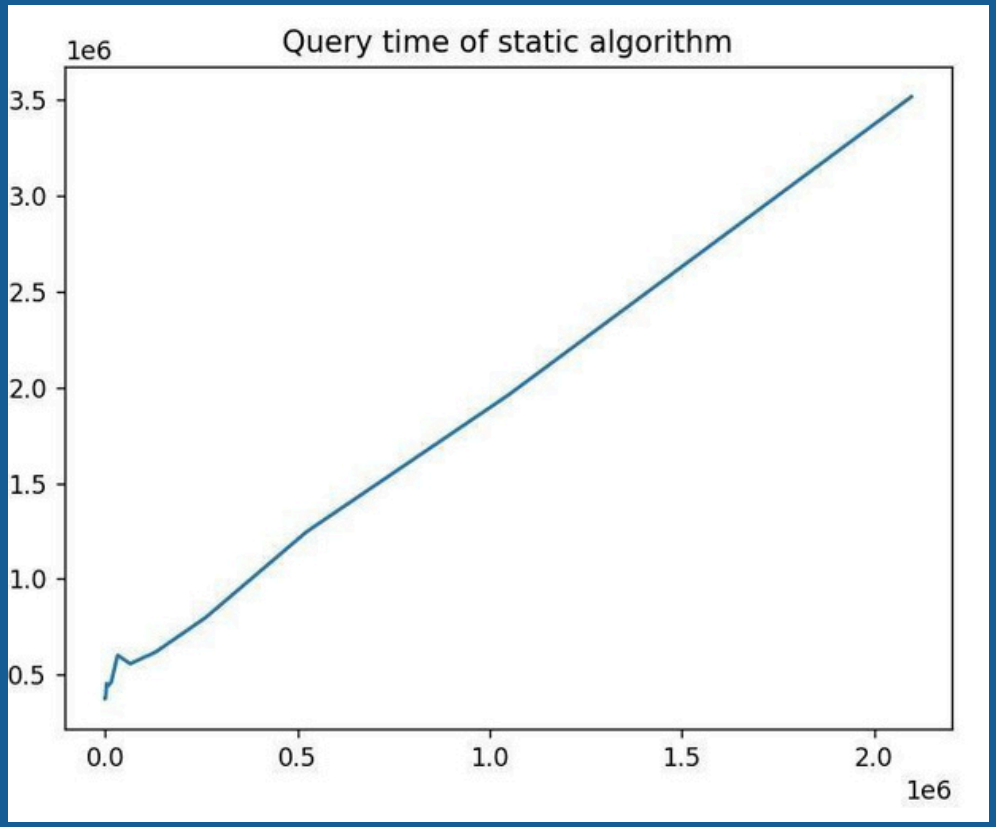
At the time of insertion of an edge we charge $\log^2 n$ tokens to it as well. The cost of reducing level of tree and non tree edges as part of delete operations takes k*log(n) time but we will account for this using the tokens stored up in the insertion phase, that is reducing each edge's token balance of $\log^2(n)$ by a factor of log(n). Hence the amortised costs are as follows:

Insert : log(n) (edge insertion) + $\log^2(n)$ (token balance)

Delete : $\log^2(n)$ (edge deletion from corresponding spanning forests) + 0 (edge level reduction)

Hence the overall amortised running time of an insert or delete operation is $O(\log^2(n))$.

PLOTS



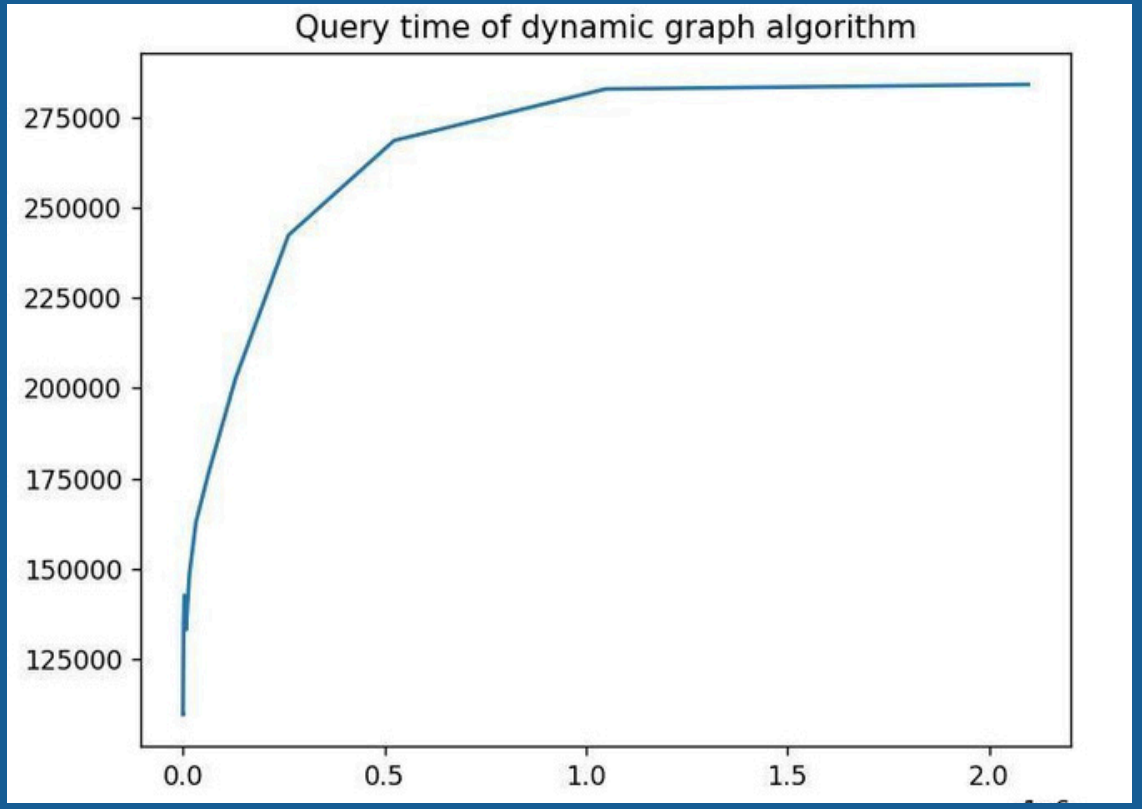
Key Observations:

- Linear Growth: The algorithm exhibits an approximately linear relationship between query size and time, suggesting it scales well for increasing workloads.
- Small Overhead: At smaller input sizes, the query time begins with slight variations but stabilizes as the input increases.
- Performance Trend: The consistent slope indicates steady efficiency with no sudden computational bottlenecks.

PLOTS

Key Observations:

- Rapid Growth Phase: The query time rises steeply for smaller input sizes, indicating initial setup or processing costs.
- Stabilization: After reaching a peak, the query time plateaus, suggesting efficient handling of larger inputs without significant degradation in performance.
- Performance Ceiling: The curve flattens, implying the algorithm maintains a consistent upper limit on query time for higher input sizes.



THANK YOU