Dynamic Graph Algorithms

CSC-531

Incremental All Pairs Connectivity

Dr. Shahbaz Khan
Department of Computer Science and Engineering,
Indian Institute of Technology Roorkee
shahbaz.khan@cs.iitr.ac.in

Incremental Connectivity

Single Source Connectivity:

Given an undirected graph G(V,E) and a source vertex **s**.

Update(x,y): Add edge (x,y) to the graph G.

Query(x): Is the given vertex x connected to s in G.

WC = O(m)
Am = O(1)
O(n)

Maintaining Components of a Graph solves larger problem,
All Pairs Connectivity

All Pairs Connectivity:

Given an undirected graph G(V,E).

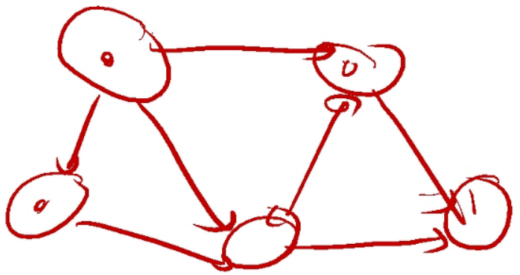
Update(x,y): Add edge (x,y) to the graph G.

Query(x,y): Is the given vertex x connected to y in G.

WC = O(mn)
Am = O(1)

General Graph

Super vertex for SCC



DAG

All pair Reach in Static setting is O(mn)

Undirected — Components BFS/DFS O(m+n)

Incremental Connectivity

All Pairs Connectivity:

Incremental Connectivity

All Pairs Connectivity:

Given an undirected graph G(V,E).

Update(x,y): Add edge (x,y) to the graph G.

Query(x,y): Is the given vertex x connected to y in G.

- **Algorithm:**

SIMPLE ALGORITHM

Use Lists and Arrays to get a simple algorithm

Worst Case Time O(n)
Amortized time O(log n)

Incremental Connectivity

All Pairs Connectivity:

Given an undirected graph G(V,E).

Update(x,y): Add edge (x,y) to the graph G.

Query(x,y): Is the given vertex x connected to y in G.

- **Algorithm:**
  Store all elements in a component ci, in list L[i]
  Store component index for every element in Array A[i]

L[#c] = { | }

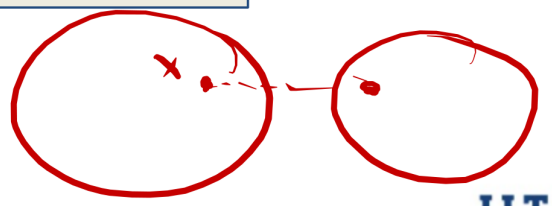
A[n]

Query(x,y): return A[x] == A[y]

Update(x,y): L[i] ∪ L[j] = L[k] x L[l]

A[.] = k

(x, y)



Incremental Connectivity

All Pairs Connectivity:

Given an undirected graph G(V,E).

Update(x,y): Add edge (x,y) to the graph G.

Query(x,y): Is the given vertex x connected to y in G.

- **Algorithm:**

  Store all elements in a component ci, in list L[i]

  Store component index for every element in Array A[i]

Query(x): Report A[x]==A[s]

Update(x,y): If(A[x]!= A[y])

Merge L[A[x]] and L[A[y]]

Update Index of elements in L[A[x]]

Incremental Connectivity

All Pairs Connectivity:

Given an undirected graph G(V,E).

Update(x,y): Add edge (x,y) to the graph G.

Query(x,y): Is the given vertex x connected to y in G.

- **Algorithm:**
  Store all elements in a component ci, in list L[i]
  Store component index for every element in Array A[i]

Query(x): Report A[x]==A[s]

O(1) WC Time

Update(x,y): If(A[x]!= A[y])
Merge L[A[x]] and L[A[y]]
Update Index of elements in L[A[x]]

O(n) WC Time

Amortized?

WC Example

i^{th} ins takes $\Omega(i)$

total $\Omega(n^2)$

Am. $\Omega(n)$

Incremental Connectivity

All Pairs Connectivity:

Given an undirected graph G(V,E).

Update(x,y): Add edge (x,y) to the graph G.

Query(x,y): Is the given vertex x connected to y in G.

- **Algorithm:**

  Store all elements in a component ci, in list L[i]

  Store component index for every element in Array A[i]

Query(x): Report A[x]==A[s] — **O(1)** WC Time

Update(x,y): If(A[x]!= A[y])
Merge L[A[x]] and L[A[y]]
Update Index of elements in L[A[x]] — **O(n)** WC Time

Can be made *O(log n)* by updating *smaller* list — Amortized **O(n)**

Incremental Connectivity

All Pairs Connectivity:

- **Algorithm:**
  Store all elements in a component ci, in list L[i]
  Store component index for every element in Array A[i]

Query(x): Report A[x]==A[s]

O(1) WC Time

Update(x,y): If(A[x]!= A[y])
Merge L[A[x]] and L[A[y]]
Update Index of elements in
smaller of L[A[x]] or L[A[y]]

O(n) WC Time

Amortized **O(log n)**

How many times can A[x] for an element x update?

$|A| \leq |B|$

$|A|+|A| \leq |A|+|B|$

$|A|$

x

$|A|+|B| \geq 2|A|$

Incremental Connectivity

All Pairs Connectivity:

- **Algorithm:**
  Store all elements in a component ci, in list L[i]
  Store component index for every element in Array A[i]

Query(x): Report A[x]==A[s]

O(1) WC Time

Update(x,y): If(A[x]!= A[y])
Merge L[A[x]] and L[A[y]] → O(1)
Update Index of elements in
smaller of L[A[x]] or L[A[y]]

O(n) WC Time

Amortized **O(log n)**

How many times can A[x] for an element x update?

When is A[x] for an element x updated? => Size of new L[A[x]] doubles.

=> Number of doubling possible <= log n

Total Update time O(nlog n)

Number of updates?

Am = O(n)/n = O(1)

Incremental Connectivity

All Pairs Connectivity:

- **Algorithm:**
  Store all elements in a component ci, in list L[i]
  Store component index for every element in Array A[i]

Query(x): Report A[x]==A[s]

O(1) WC Time

Update(x,y): If(A[x]!= A[y])
Merge L[A[x]] and L[A[y]]
Update Index of elements in
smaller of L[A[x]] or L[A[y]]

O(n) WC Time

Amortized **O(log n)**
Total Update time: O(nlog n)
Number of updates: $n' \rightarrow \Omega(n-1)$ updates
n' updates in O(n'log n') time
Amortized time= O(log n') = O(log n)

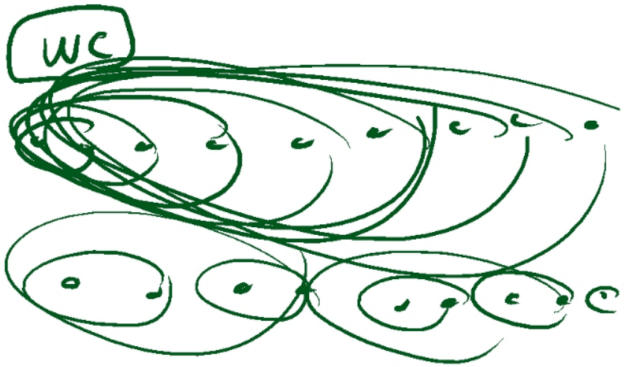
PROOF:
Consider any element e in a set of size N'
Number of Unions = N'-1
Say union of A and B,
|A|-1 + |B|-1 +1 = |A|+|B|-1
Number of Unions affecting e < log N'
Total = O(N' log N')

Incremental Connectivity

All Pairs Connectivity:

$Dm_2 \quad \frac{N' \log N'}{N'-1} \leq \frac{2N' \log n}{N}$

$N' > 1 \quad N'-1 \geq N'/2$



Phase 2^i $n/2$

Total $= \Omega(n \log n)$

Terminator $A[i] = n'$

Incremental Connectivity

All Pairs Connectivity:

- **Algorithm:**
  Store all elements in a component c_i in list L[i]
  Store component index for every element in Array A[i]

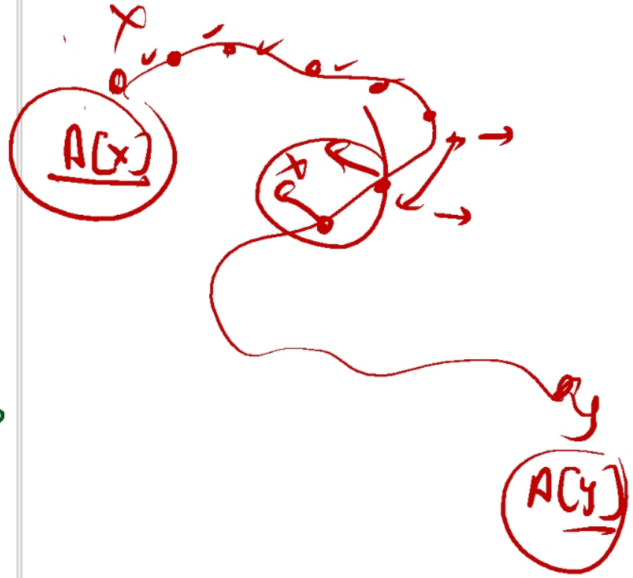
Query(x): Report A[x]==A[s]

Update(x,y): If(A[x]!= A[y])
Merge L[A[x]] and L[A[y]]
Update Index of elements in
smaller of L[A[x]] or L[A[y]]

O(1) WC Time

O(n) WC Time

Correctness:
x and y are connected <=> A[x] = A[y]



Incremental Connectivity

All Pairs Connectivity:
Correctness:
x and y are connected=>A[x]=A[y]

Consider a path from x to y :
x->a->b->c…->p->q->r->y

Assume for contradiction
A[x] != A[y]

Query(x): Report A[x]==A[s]

Update(x,y): If(A[x]!= A[y])
Merge L[A[x]] and L[A[y]]
Update Index of elements in
smaller of L[A[x]] or L[A[y]]

Then for some edge along the path (i,j) A[i] != A[j]

Which is not possible when the edge was processed, it would Merge.

Hence, assumption is wrong and A[x]=A[y]

Incremental Connectivity

All Pairs Connectivity:

Correctness:

A[x]=A[y] => x and y are connected

Consider the insertion (i,j) when A[x] became equal to A[y].

x --- (i,j) --- y

Query(x): Report A[x]==A[s]

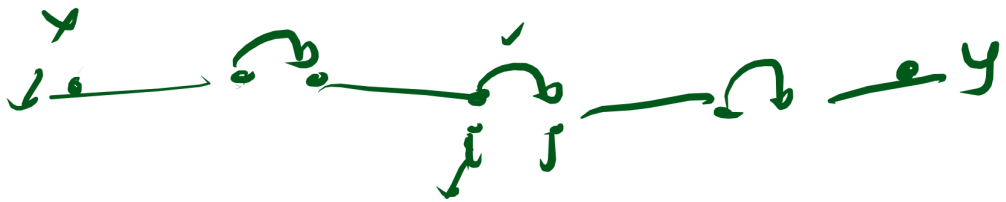
Update(x,y): If(A[x]!= A[y])
Merge L[A[x]] and L[A[y]]
Update Index of elements in ***smaller*** of L[A[x]] or L[A[y]]

Similarly, Consider insertion when A[x] became equal to A[i], and A[j] with A[y]

x -(a,b)- i-j -(c,d)- y

Continue until all edges of the path revealed.

We get x is connected to y by using the computed path.



Disjoint Set Union

Operations:

- ***Find(x)*** = Report the set containing x
- ***Union(x,y)*** = Merge the sets containing x and y

Implementation:

Linked List:

Store all elements in Set Si, in list L[i]

Store set index for every element in Array A[i]

Find(x) = Report A[x] **O(1) Time**

Union(x,y) = Merge L[A[x]] and L[A[y]] **O(1) Time**

Update Index of elements in L[A[x]] **O(n) Time**

Without Smaller Set Heuristic **O(n)**

With Smaller Set Heuristic **O(log n)**

Disjoint Set Union

Operations:

- ***Find(x)*** = Report the set containing x
- ***Union(x,y)*** = Merge the sets containing x and y

Implementation:

Linked List:

Store all elements in Set Si, in list L[i]
Store set index for every element in Array A[i]

Find(x) = Report A[x] **O(1) Time**
Union(x,y) = Merge L[A[x]] and L[A[y]] **O(1) Time**
Update Index of elements in L[A[x]] **O(n) Time**

Without Smaller Set Heuristic **O(n)**
With Smaller Set Heuristic **O(log n)**

Tree:

Store all elements in Set Si, in a Tree T[i]
Store pointer to parent in Tree

Find(x) = Report Root(x)
Union(x,y) = Find(x) and Find(y)
Make Root of ***smaller*** tree a child of the root of ***larger*** tree

With Smaller Tree Heuristic O(log n)
With Smaller Rank Heuristic O(log n)
With Rank + Path Compression O(log* n)

Disjoint Set Union

Operations:

- ***Find(x)*** = Report the set containing x
- ***Union(x,y)*** = Merge the sets containing x and y

Implementation:

Tree:

Store all elements in Set Si, in a Tree T[i]

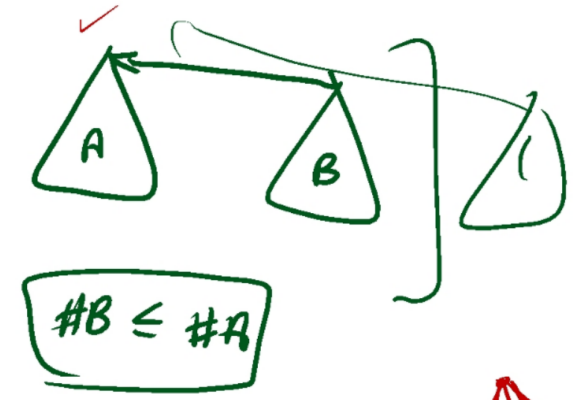
Store pointer to parent in Tree

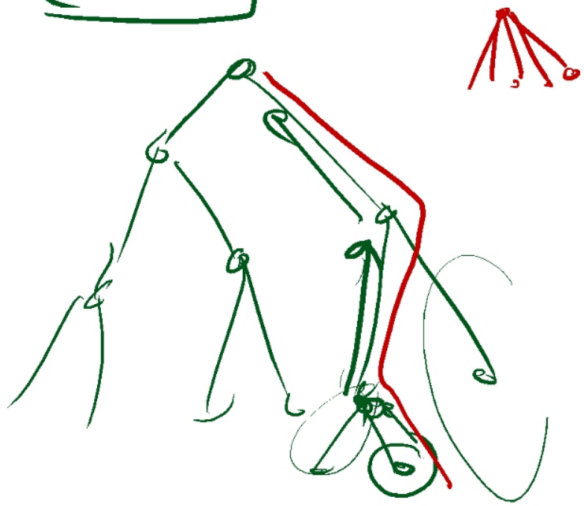
Find(x) Report Root(x)

Union(x,y) Find(x) and Find(y). Make Root of ***smaller*** tree a child of the root of ***larger*** tree

With Smaller Tree Heuristic (Number of elements) O(log n)

Ω(log n)





Disjoint Set Union

Operations:

- ***Find(x)*** = Report the set containing x
- ***Union(x,y)*** = Merge the sets containing x and y

Implementation:

Tree:

Store all elements in Set Si, in a Tree T[i]

Store pointer to parent in Tree

Find(x) Report Root(x)

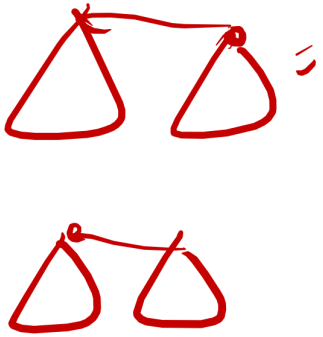
Union(x,y) Find(x) and Find(y). Make Root of ***smaller*** tree a child of the root of ***larger*** tree

With Smaller Tree Heuristic (Height) $=$ O(log n) $=$

Disjoint Set Union

Operations:

- ***Find(x)*** = Report the set containing x
- ***Union(x,y)*** = Merge the sets containing x and y



Implementation:

Tree:

Store all elements in Set Si, in a Tree T[i]

Store pointer to parent in Tree

Find(x) Report Root(x)

Union(x,y) Find(x) and Find(y). Make Root of ***smaller*** tree a child of the root of ***larger*** tree

WIth Smaller Rank Heuristic

1- *Rank(single element)* = *0*

2- Union T(r1) and T(r2)

If (Rank(r1)>= Rank(r2))

Make r1 as root

Rank(r1)= max(Rank(r1),Rank(r2)+1)

Disjoint Set Union

Operations:

- ***Find(x)*** = Report the set containing x
- ***Union(x,y)*** = Merge the sets containing x and y

Implementation:

Tree:

Store all elements in Set Si, in a Tree T[i]

Store pointer to parent in Tree

Find(x) Report Root(x)

Union(x,y) Find(x) and Find(y). Make Root of ***smaller*** tree a child of the root of ***larger*** tree

WIth Smaller Rank Heuristic

=> Height

O(log n)

1- *Rank(single element) = 0*

2- Union T(r1) and T(r2)

If (Rank(r1)>= Rank(r2))

Make r1 as root

Rank(r1)= max(Rank(r1),Rank(r2)+1)

Disjoint Set Union

WIth Smaller Rank Heuristic

Property 1: If Rank(k) = r
=> k has 2^r descendants

Property 2: Rank(k) < Rank (par(k))

Property 3: Number of vertices with Rank r = n/2^r

group (k) = log*(rank (k))

log*n = { 0 if n<=1
1+ log*(log n) if n>1

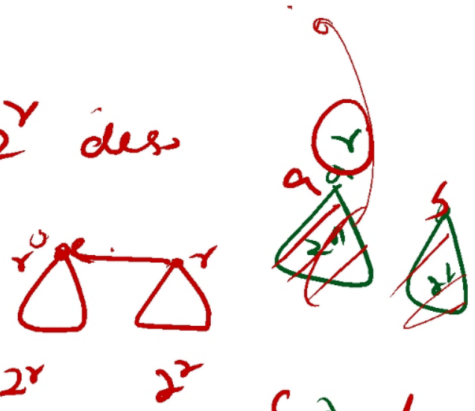
Property 1: Number of Groups => log* n

Property 2: Number of Vertices in Group g

1- *Rank(single element) = 0*
2- Union T(r1) and T(r2)
If (Rank(r1)>= Rank(r2))
Make r1 as root
Rank(r1)= max(Rank(r1),Rank(r2)+1)

Group	Rank
0	[0,1]
1	(1,2]
2	(2,2^2]
3	(2^2,2^2^2]
g	(2^^g-1, 2^^g]

$\geq 2^r$ des



$r \to r+1$

$2^r \to 2^{r+1}$

V_a V_b V_c V_d V_e V_f

Property 3

When $x \in V$ became $rank(x) = r$

it had $\geq 2^r$ descendants

It can only be taken away by some vertex who is ancestor of x having larger rank

Among all vertices with rank r

x is uniquely associated with $\geq 2^r$ vertices

elem with rank $r \leq n/2^r$

$a + ap + ap^2 + \cdots + ap^k$

$a\left(\frac{p^{k+1}-1}{p-1}\right)$ $\quad$ $a\left(\frac{1-p^{2k+1}}{1-p}\right)$

$\frac{n}{2^5}\left(\frac{1-(1/2)^4}{1/2}\right)$

$\frac{2n}{2^{nn}g}$

$\frac{n}{2^{(2^{nn}(g-1))}}\left(\frac{1-(1/2)^{2^{nn}g-2^{nn}(g-1)}}{1/2}\right)$

vertices in graph g

vertices with rank

$2^{nn}g-1$ to $2^{nn}g$

$\sum_{r=2^{nn}(g-1)+1}^{2^{nn}g} \left[\frac{n}{2^r}\right]$

$\frac{n}{2^{2^{nn}(g-1)}} + \frac{n}{2^{2^{nn}g}} \cdots \frac{n}{2^{2^{nn}g}}$

$\frac{n}{2^5} + \frac{n}{2^6} + \frac{n}{2^7} + \frac{n}{2^8}$

Disjoint Set Union

Path Compression:

After Find(x), Make Root(x) the parent of each element on the path.

→ Rank not updated in PC

→ Root does not have change in descendents in PC

Algorithm:

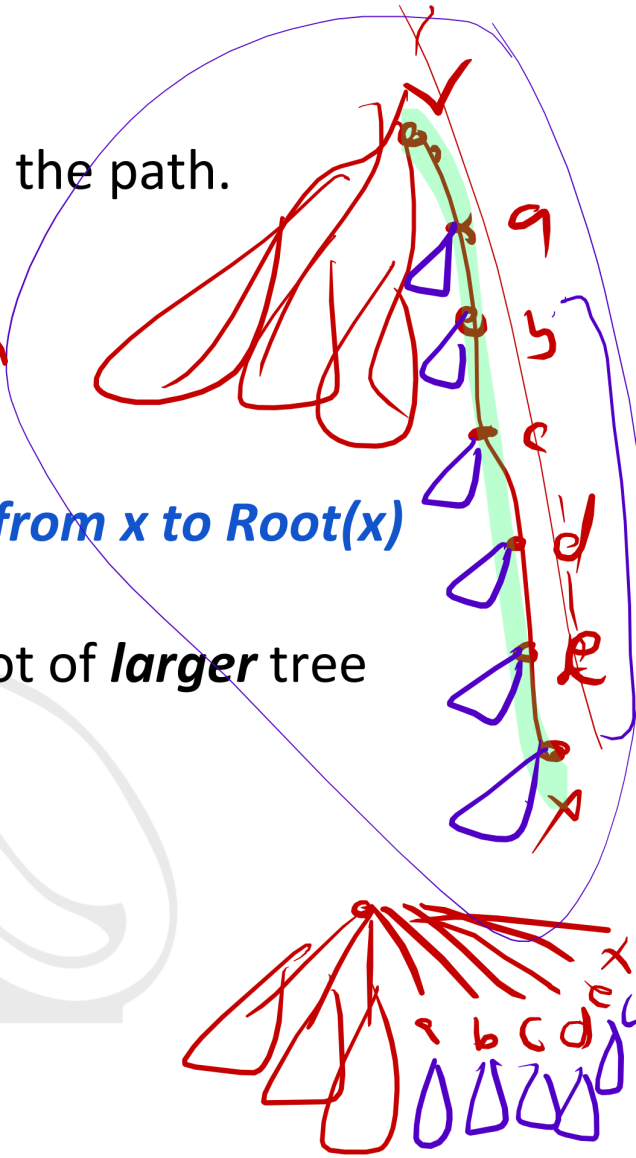
Find(x) Report Root(x). ***Path Compression on path from x to Root(x)***

Union(x,y) **a** = Find(x) and **b** = Find(y).

Make Root of ***smaller*** tree a child of the root of ***larger*** tree

```
If(Rank[a] >= Rank[b])
    Make b the child of a
    Rank[a] = max(Rank[a], Rank[b]+1)
else
    Make a the child of b
    Rank[b] = max(Rank[b], Rank[a]+1)
```

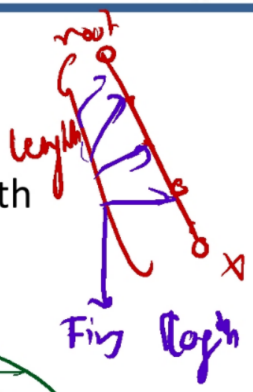
O(1)



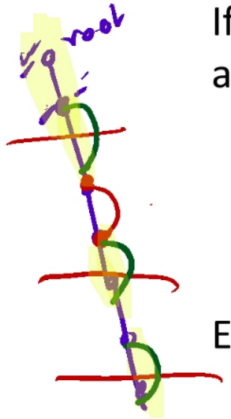
Disjoint Set Union

Analysis: COST of FIND(x) is length of root path of x =
Distance of x from root

Charge this cost of distance on the operation and elements in the path



For each element u in path from x to root



If parent of u changes [has a grandparent]
and group(u) = group(parent(u))

charge 1 to u

Charge to vertex

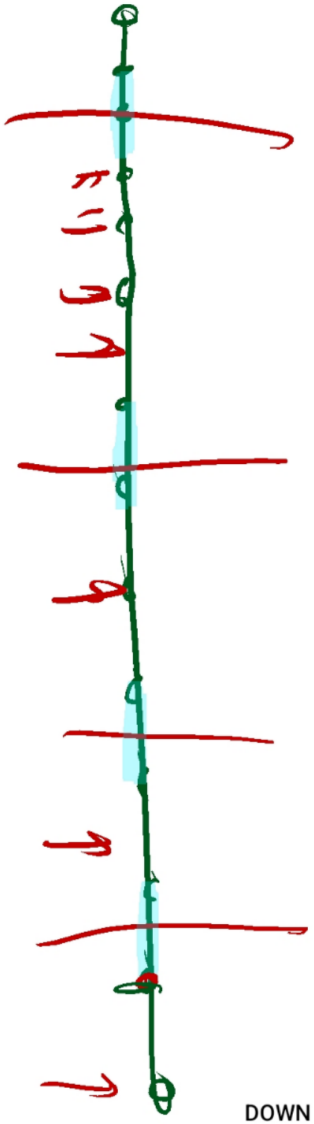
Else group(u) ≠ group(Par(u))

charge 1 to FIND(x)

#groups ≤ log n

Charging on find(x)

FIND Operation completely charged to operation or elements



Disjoint Set Union

Analysis: COST of FIND(x) is length of root path of x =
Distance of x from root
Charge this cost of distance on the operation and elements in the path

For each element u in path from x to root

If parent of u changes [has a grandparent]
and group(u) = group(parent(u)) charge 1 to u

Charged only when rank of parent changes and parent rank in same group <= 2^^group(u)

Else charge 1 to FIND(x)

[Charged for root and child of root + Number of group changes from x to root] <= 1 + log*n

FIND Operation completely charged to operation or elements

Disjoint Set Union

Analysis: COST of FIND(x) is length of root path of x =
Distance of x from root

Charge this cost of distance on the operation and elements in the path

For each element u in path from x to root

If parent of u changes [has a grandparent]
and group(u) = group(parent(u)) charge 1 to u

Charged only when rank of parent changes
and parent rank in same group <= 2^^group(u)
Total Cost = Sum of Charge on each element
= Sum of Charge on each group = n log*n

Else charge 1 to FIND(x)

[Charged for root and child of root +
Number of group changes from x to root] <= 1 + log*n

FIND Operation completely charged to operation or elements

Disjoint Set Union

Summary:

1- Only expensive operation is FIND(x) costing length of x to root path

2- Charged on operation and elements such that

(a) Parent's changed

(b) Rank of parent in same group

(c) Making total charge equal to number of ranks in the group (2^^g)

3- Total charge on each group g = Number of elements x Charge on element

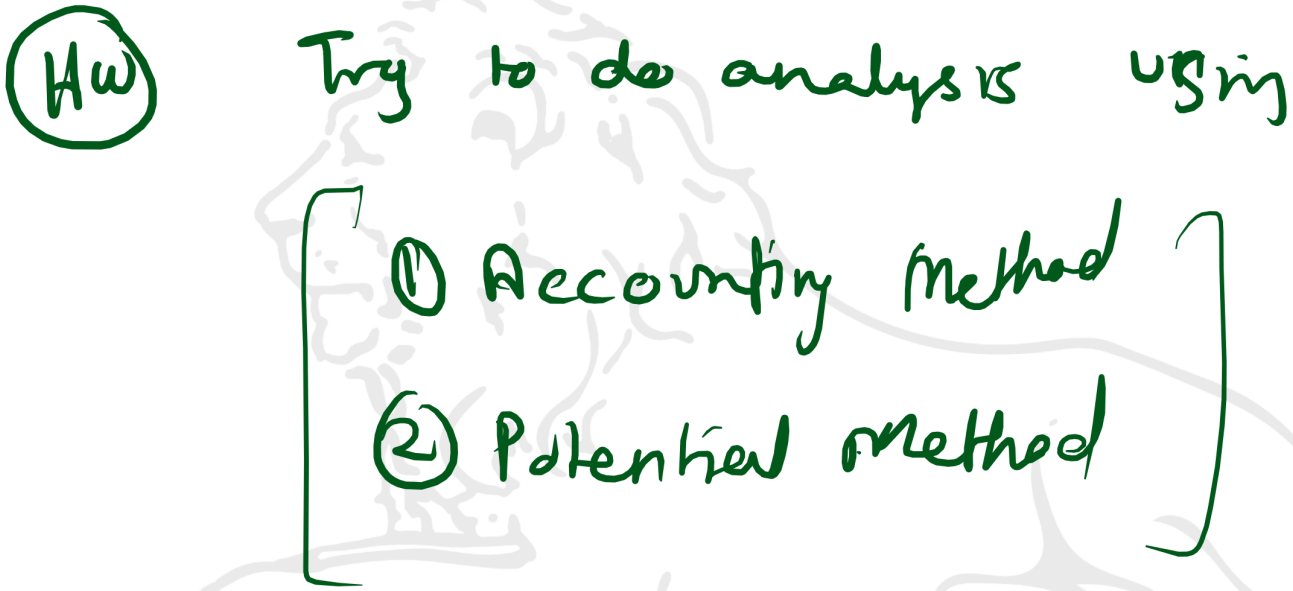
= n/2^^g x 2^^g = n

4- Charge on FIND(x) = O(log*n) and All groups O(nlog*n) => Total O(nlog*n)

5- Charging on elements such that it can be associated to size of group.

Disjoint Set Union

Reference: https://cs.uwaterloo.ca/~r5olivei/courses/2020-fall-cs466/lecture1.pdf



Disjoint Set Union

Analysis: COST of FIND(x) is length of root path of x =
Distance of x from root
Charge this cost of distance on the operation and elements in the path

For each element u in path from x to root

If parent of u changes [has a grandparent]
and group(u) = group(parent(u)) charge 1 to u

Charged only when rank of parent changes
and parent rank in same group <= 2^^group(u)
Total Cost = Sum of Charge on each element
= Sum of Charge on each group = n log*n

Else charge 1 to FIND(x)

[Charged for root and child of root +
Number of group changes from x to root] <= 1 + log*n

FIND Operation completely charged to operation or elements

$$\sum_{x \in V} \text{charge}(x)$$

$$= \sum_{g \in G} \left[\sum_{g(x)=g} \text{charge}(x) \right]$$

$$= \sum_{g \in G} \frac{n}{2^{\wedge\wedge g}} \times 2^{\wedge\wedge g}$$

$$= n \times \log^* n$$