Minimum Cost Spanning Tree

$G = (V, E)$: Undirected graph

Tree: $T = (V', E')$; $V' = V$; $|E'| = |V| - 1$

↓

induced by the graph G

T has no cycles

$T - e + e'$ is also a Tree

↓ $\in E'$ ↓ $\in E$

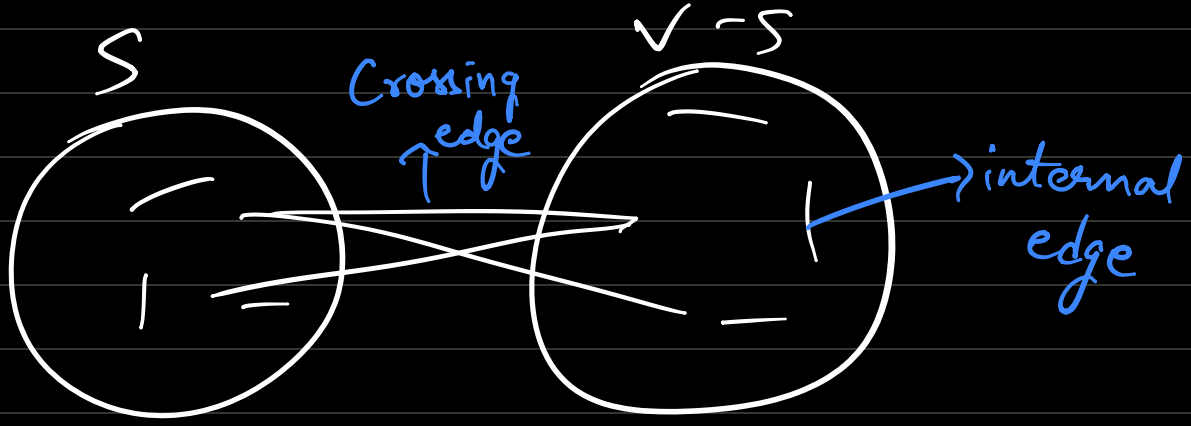
Vertex Set partition in a Graph:

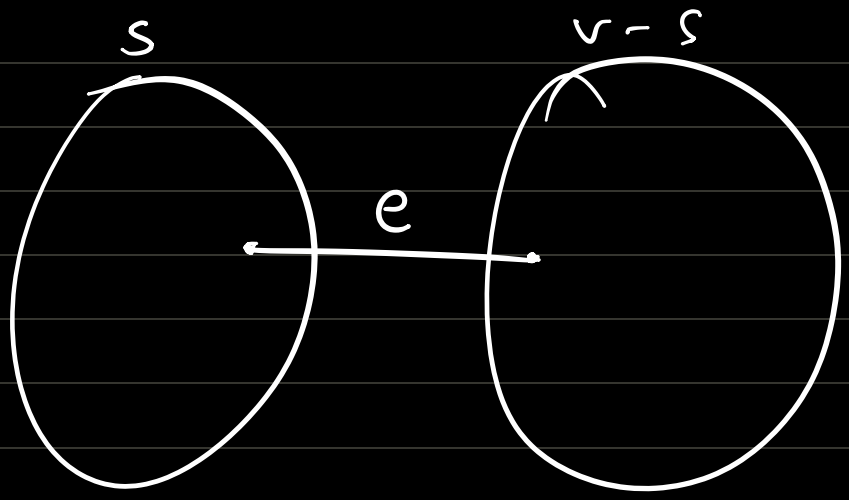
$(S, V-S)$ is a partition and

Internal edge => (u, v) and u and v, both belong to exactly one of the partitions

Crossing edge => (u, v) and $u \in S$ and $v \in V-S$ (or) $u \in V-S$ and $v \in S$

Let $A \subseteq E$. We say a cut $(S, V-S)$ is respecting A iff all the edges in A are internal w.r.t the cut, $(S, V-S)$.





Let e be a crossing edge and e be of minimum weight among the crossing edges existing w.r.t cut (S, V-S).

$\Rightarrow$ $A \cup \{e\} \subseteq$ some Spanning tree

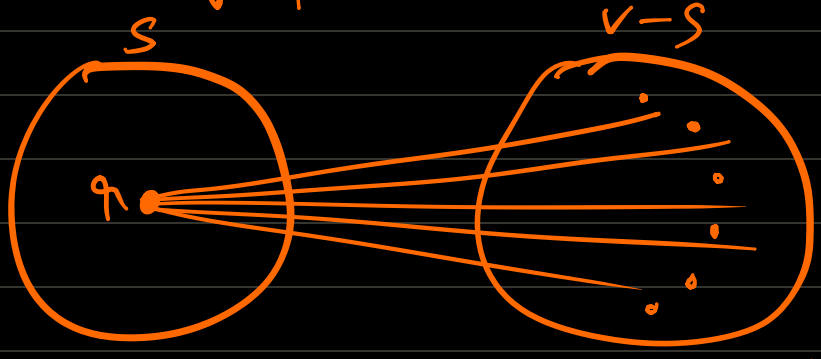
$A \subseteq$ Some Spanning Tree

e is a min. weight crossing edge of a cut respecting A.

Then, $A \cup \{e\} \subseteq$ Some Spanning Tree

$A : \phi$ (initialization)

So, initially, the cut can have any vertex in the graph in the vertex set S.



$S = \{q\}$ $\quad V - S = V - \{q\}$

Look at the adjList (q) and get the minimum edge weight

min. Cost Crossing edge is an edge (u, v) Such that

$$w(u, v) \leq w(u, x) \quad \forall x \in adjList(u)$$

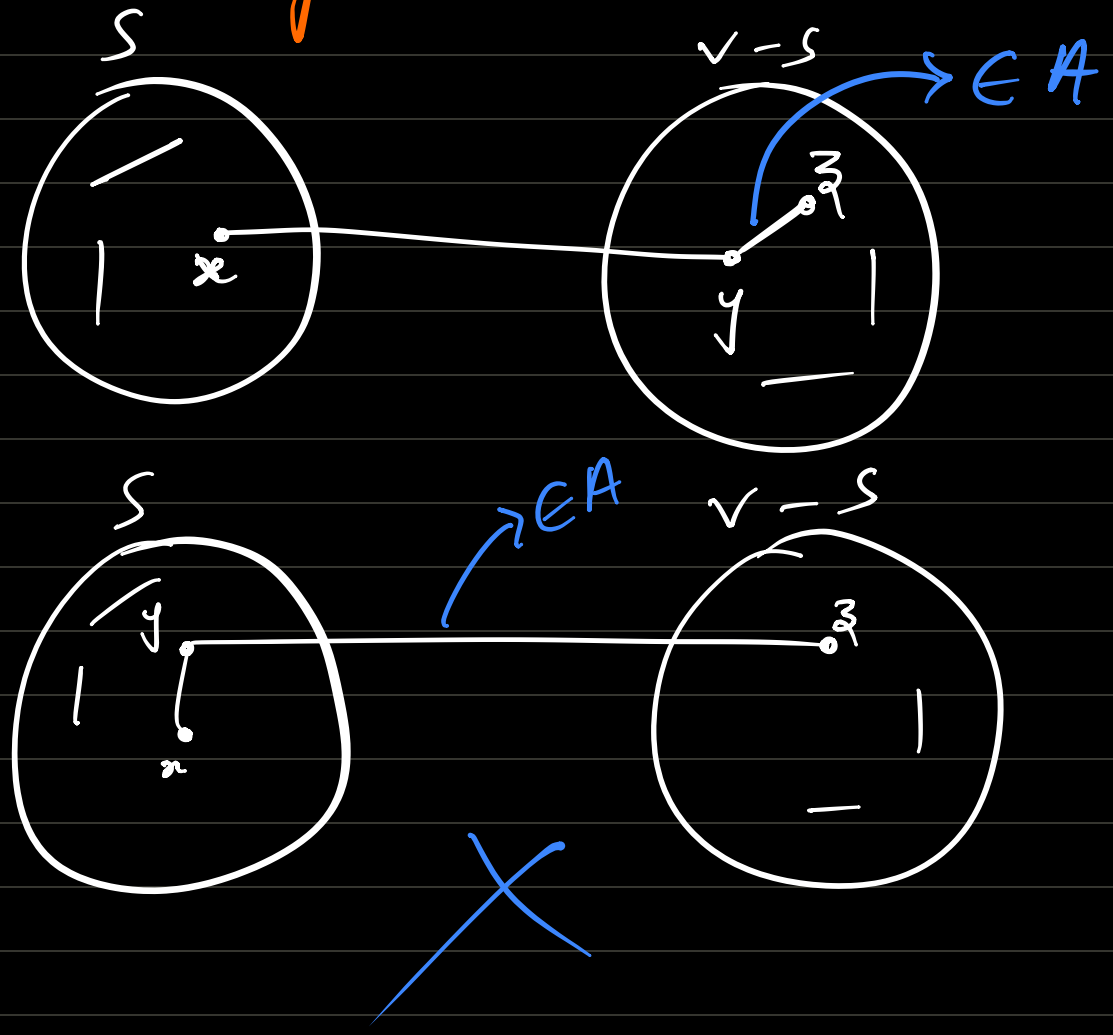
Update : $A \leftarrow A \cup \{(u, v)\} = \phi \cup \{(u, v)\}$

$= \{(u, v)\}$

Move : $S \leftarrow S \cup \{v\}$

$V - S \leftarrow V - S$

✗ an internal edge will become a crossing edge when we move v to S.



Idea: Make Sure that no edge in A is in V - S.

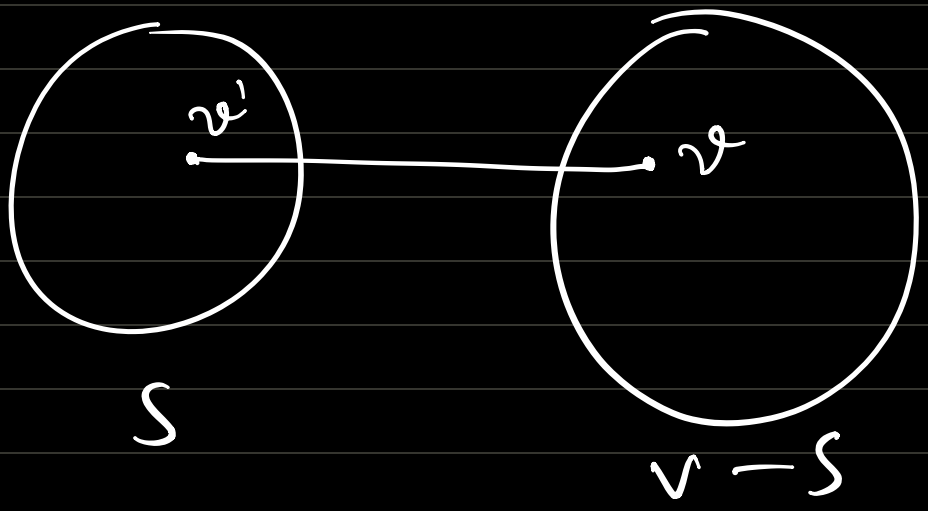
i.e, we maintain all edges of A as internal to S.

Development of Prims Algorithm:

S	$V-S$
$\{s\}$	$V-\{s\}$

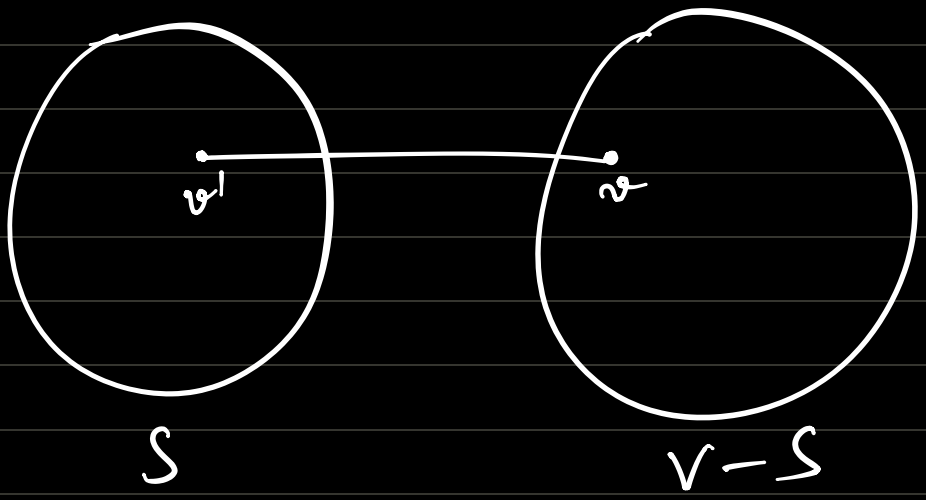
* for every $v \in V-S$, we maintain a minimum weight crossing edge (v, v') such that $v' \in S$.

Then, the overall min. weight crossing edge can be identified by only looking at the type of edges (v, v') $\forall$ $v \in V-S$.



$\Rightarrow$ By examining only $|V-S|$ edges, we identify the minimum crossing edge

$\therefore$ $|V-S| \leq n-1$, this is a significant improvement.



(v, v') : min. weight crossing edge incident on v.

$\Rightarrow \quad w(v, v') \leq w(v, u) \quad \forall \; u \in S$

i.e, $\forall$ crossing edges (v, u)

$v' = p(v)$ (partner of v)

$c(v) = w(v, p(v))$

∴ for each $v \in V - S$, we maintain $p(v)$ and $c(v)$ arrays.

Let $v \in V - S$ such that

$$c(v) \leq c(u) \quad \forall \; u \in V - S$$

Then, clearly $(v, p(v'))$ is the min. wt. crossing edge.

Hence, add $(v, p(v))$ to A
move v to S and $p(u)$, $c(u)$ values must be updated $\forall u \in V - S$

$p(u)$ is updated only if $\exists (v,u) \in E$ and $w(v,u) < w(u, p(u))$

$p(u) \leftarrow v$.

High Level pseudocode

$A = \phi$

$V - S = V - \{r\}$

For all $v \in V$,

$p(v) = \text{NULL}$

$c(v) = \infty$

For each $u \in V - S$

if $(r, u) \in E$

$p(u) = r;$

$c(u) = w(r, u);$

while $(V - S \neq \phi)$

{

Find a vertex $v \in V - S$ with minimum $c(v)$

move v to S and add $(v, p(v))$ to A

Update $c(u)$ and $p(u)$ $\forall$ $u \in V - S$

}

Detailed Pseudocode for Prim's MST Algorithm

```
primMST((V,E))        // undirected graph, remember!
{
  S = Φ
  p(v) = NULL   ∀ v ∈ V
  c(v) = ∞      ∀ v ∈ V

  S = {r}

  For all u ∈ V−S
  {
    if (u,r) ∈ E
      p(u) = r
      c(u) = w(u,r)
  }

  while (V−S ≠ Φ)
  {
    1) Find v ∈ V−S such that
       c(v) ≤ c(u)  ∀ u ∈ V−S

    2) move v to S

    3) For each u ∈ V−S
       if (u,v) ∈ E && w(u,v) < c(u)
         p(u) = v
         c(u) = w(u,v)
  }

  Return p(v)
}

// T = (V, A) is an MST
// A = {(v, p(v)), v ∈ V−{r}}
```

Kruskal's Algorithm for MST

Kruskal's Algorithm:

→ Greedy algorithm.

- only select those edges which contribute to the final MST at any time-step in the algorithm.

→ **Rejection Rule:**
we reject an edge $e \in E$ if it forms a cycle at any point in the algorithm

Outline

// $G = (V, E)$: Connected, weighted, undirected graph

// $T = (V, A)$: Output MST, a tree with $n-1$ edges.

// A : set of edges selected by the algorithm.

// L : list of sorted edges according to their edge weights in non-decreasing order.

1. $A = \phi$; e = first edge of L

2. while (NOT end(L))

   2.1. if (e does not form a cycle with any subset of edges in A)
   $A \leftarrow A \cup \{e\}$

   2.2. e = next(e, L)

A : acyclic, connected graph

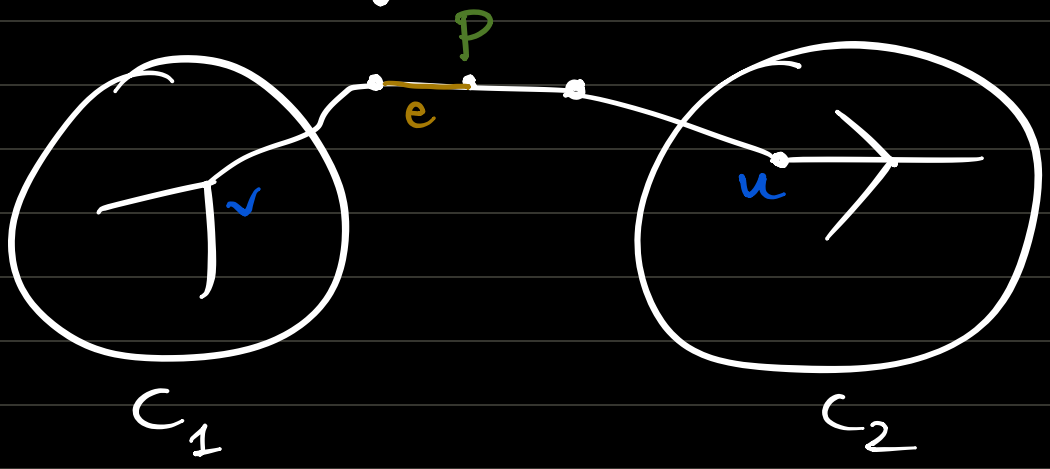
Proving the Correctness of Kruskal's Algorithm for MST:

T = (V, A) is a Tree

To prove: ① set of edges in A form a tree
② The graph induced by A is Connected.

P : a path in G connecting 2 diff. vertices belonging to 2 diff. components

e : an edge in P ; $e \notin A$



* Not all edges in P are in A. Otherwise, C_1 and C_2 won't be 2 diff. Components.

* Let e be the 1st edge that is not in A ↳ $e \in P$

* This will not form a cycle with any set of edges in A.

* If it forms, it would define a path to a vertex outside this Component

* $e \notin A$ is a Contradiction because when e was examined by the algorithm, e would have been included in A.

Thus, A induces a connected acyclic graph and this means A forms a tree.
Thus, A contains exactly $(n-1)$ edges, where $n = |V|$

∴ We've Shown T is a Tree.

Let $x_1 < x_2 < \dots < x_{n-1}$ $\forall x_i \in A$ & $1 \le i \le n-1$

$\forall e_1, e_2 \in E$, $w(e_1) \neq w(e_2)$ ⇓ all edge weights are distinct

x_i ↓ edge in A ↓ set of edges in the final MST produced by Kruskal's algorithm.

If T is not a MST let T' be a MST with $w(T') < w(T)$

and let $y_1 < y_2 < \ldots < y_{n-1}$ $\forall y_i \in$

algorithm's o/p: $x_1 < x_2 < \ldots < x_j \leq x_{j+1} < \ldots < x_{n-1}$

$\; \| \;\;\;\;\; \| \;\;\;\;\;\;\;\; \| \;\;\;\; \neq$

$y_1 < y_2 < \ldots < y_j < y_{j+1} < \ldots < y_{n-1}$

j : first index in $1, 2, \ldots, n-1$ such that $x_j \neq y_j$

$\forall i < j$, $x_i = y_i$

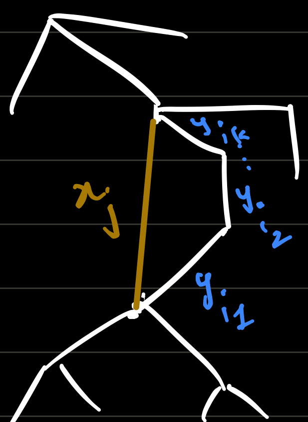
Thus, we have $w(x_j) < w(y_j)$
($\because x_j$ was chosen greedily by the algorithm)

T'' be a spanning tree cheaper than T' that we try to construct.

$\langle y_{i_1}, y_{i_2}, \ldots, y_{i_k} \rangle$: a path in T' after we add x_j to T'

$\downarrow$

This is a unique cycle C



$x_j + T'$: cycle C

If $\langle y_{i_1}, y_{i_2}, \ldots, y_{i_k} \rangle \subseteq \{y_1, y_2, \ldots, y_{j-1}\}$

then $y_{i_1} = x_{i_1}$, $y_{i_2} = x_{i_2}$, $\ldots$, $y_{i_k} = x_{i_k}$

because $\langle y_1, \ldots, y_{j-1} \rangle = \langle x_1, \ldots, x_{j-1} \rangle$

Thus, $x_j, x_{i_1}, x_{i_2}, \ldots, x_{i_t}$ would form the cycle C.

This is impossible as $x_j, x_{i_1}, x_{i_2}, \ldots, x_{i_t}$ are all edges of T and T is a tree.

Hence, the path $\langle y_{i_1}, y_{i_2}, \ldots, y_{i_t} \rangle$ must contain an edge outside $\{y_1, y_2, \ldots, y_{j-1}\}$

That means the path has an edge y_t of T' where $t \geq j$

NOTE: $w(y_t) \geq w(y_j) > w(x_j)$

Now, we define $T'' = T' + x_j - y_t$

T'' is now a tree, since the cycle is broken by removing y_t

$$\Rightarrow w(T'') = w(T' + x_j - y_t)$$
$$= w(T') + w(x_j) - w(y_t)$$
$$< w(T')$$

This contradicts the minimality of T'

Thus, a spanning tree with cost lesser than the cost of the spanning tree produced by Kruskal's algorithm cannot exist!

NOTE: Here, we're talking about the cost of the MST, and not the MST produced. There can be multiple MSTs & STs with the same total cost.

Key Question: How will you determine if e will entail a cycle?

Reframe the question:

$e = (x, y)$

current edge set of T, the MST

Check if $\exists$ path (x, y) in (V, A)

This will answer our predicate of e forming a cycle in (V, A) or not.

Connected Component:

Check if x, y belong to the Same Component of T.

$x, y \in C_1$?

NO → e doesn't form a cycle

YES → e forms cycle

Find (x): returns the name of the connected Component in T that contains x.

Let $e = (x, y)$
if $Find(x) \neq Find(y)$
    add e to A
else
    ignore e

* Adding a new edge $e=(x,y)$ to A will change the components in T:

components will reduce by 1 always when a new edge is added to A.

This operation is Union (X, Y).

↓

merges the connected components X and Y of T into a new connected component. The connected components X & Y are automatically removed.

Initialize:

$A = \phi$

e = First edge in L

initialize the names of the connected components of $T = (V, A)$

Process:

while (NOT END of L)

{

Let $e=(x,y)$

X = Find (x)

Y = Find (y)

if $(X \neq Y)$

$A \leftarrow A \cup \{e\}$

Union (X, Y)

e = next (e, L)

}

output:

$T = (V, A)$

↓

a MST of G

Now, we have to define the functions Find() and Union().

Detour: Graph Theory

Connected Component of a graph G is a maximally connected sub-graph of G.

G' : maximally connected sub-graph of G

$\Leftrightarrow$ $G' + e$ is not maximally connected for any new edge e in G and not in e'.

Acyclic $\Leftrightarrow$ no cycles

Forest = Collection of trees

↓

Connected acyclic graphs

Forest = Acyclic graph

* Relation maximally connected forms a partition on V and the relation is an equivalence relation.

Partition ADT

$V = \{1, 2, 3, \ldots, n\}$ $\quad n \geq 1$

↓ finite set of vertices

K-partition of V: Collection of K subsets of V

$S_1, S_2 \ldots, S_K$ such that

mutually exclusive → $S_i \cap S_j \neq \phi, \quad i \neq j$

$\bigcup_{i=1}^{K} S_i = V$ → mutually exhaustive

n-partition of V:

$\{1\}, \{2\}, \ldots, \{n\}$: all singletons of V.

Name of a set: Identifier of a set

∵ any vertex in V can only be in one of the partitions, we can use any element of the set to serve as an identifier of the set in a partition.

Name Array Representation

* Implementation of Find( ), Union( ) on a partition

E.g:

{1, 2, 4, 7}, {3, 8, 10}, {5, 6, 9, 11}, {12, 14}, {13}

Name: 1 3 5 12 13

Name[1] = 1
Name[2] = 1
...

Name[3] = 3
Name[8] = 3

Name[13] = 13
Name[14] = 12

Name[x] : name of the set containing x.

Defining Union( ), Find( ) operations

O(1)
```
Find(x):
    return Name[x];
```
→ global to Find & Union (here)

```
Union(X, Y):
    // w.l.o.G  assume  X < Y
    for i in range(1, ..., n):
        if (Name[i] = Y)
            Name[i] = X
```
O(n)

Name[n] : an array that stores the name of each vertex in V
n = |V|

Inverted Forest data structure

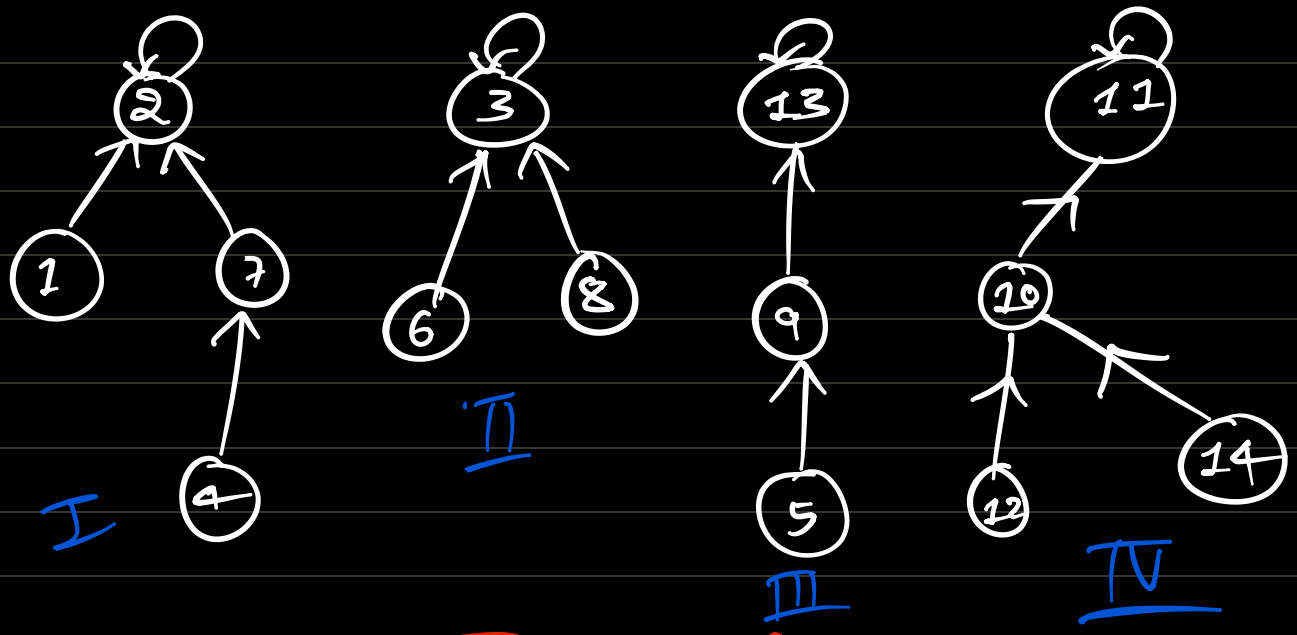
In-tree : directed rooted tree where each edge is oriented toward the root

Self-loop for the root

In-forest : Collection of intrees

A k-partition can be represented as an inforest of k in-trees.

root(Tree) = Name of the corresponding set.



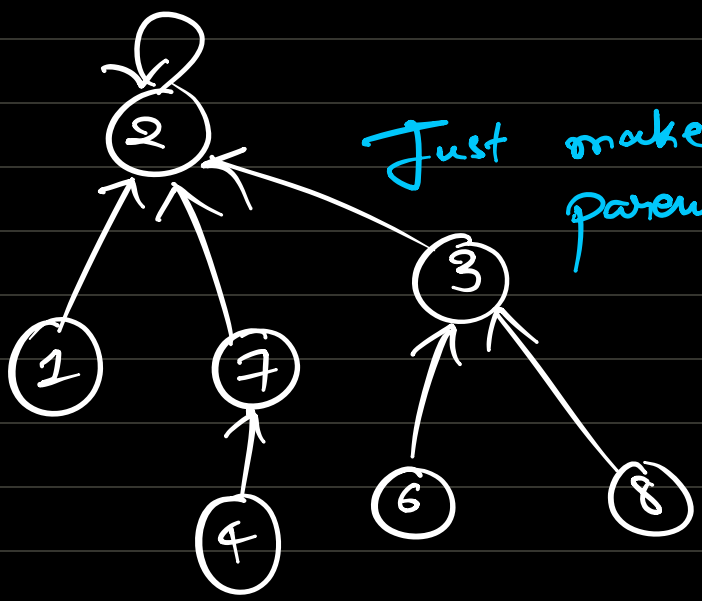
Tree Hooking

Tree hooking ≡ Union

T = Tree hook (I, II)

then T would look like

T =



p[3] = 2 //

Just make 2 as the parent of 3. //

* Find (x) is done through ancestor Chasing

Find (x)

// "ancestor chasing"

// returns the name of the set containing x.

// i.e, return the value at the root

// of the tree containing x.

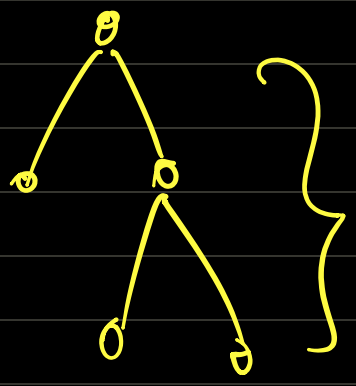
```
y = p(x)        //parent of x
while ( y ≠ p(y) )
      y = p(y)
return p(y)
```

Worst case

$\Rightarrow O(n)$

* Efficient Implementation of tree hooking would be hooking the smaller tree to the larger tree, so that the larger tree won't become any larger.

measure : height of the tree



⇓

no. of edges in the longest path from root to the leaf.

Union Algorithm

```
Union (A, B) :
    if ( size(A) ≤ size(B))
        p(A) = B
        size(B) = size(B) + size(A)
    else :
        p(B) = A
        size(A) = size(A) + size(B)
```

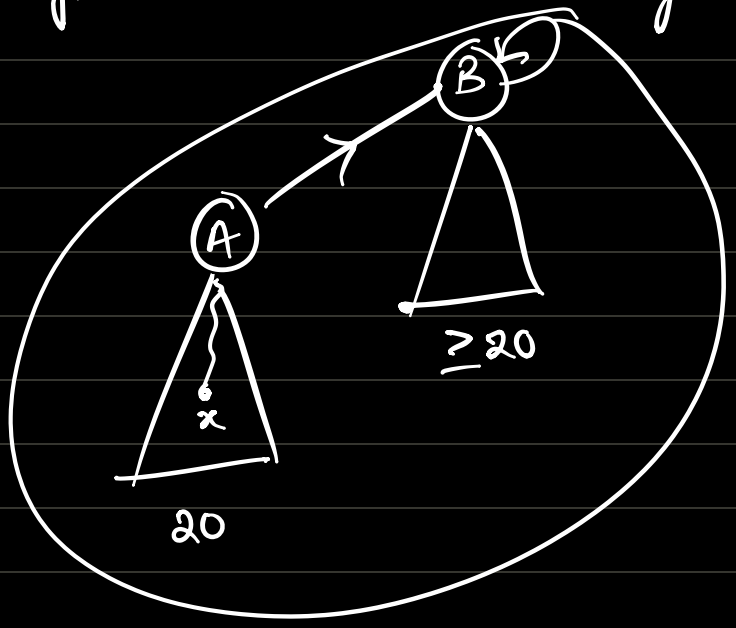
} hook smaller tree to the larger tree

Time Complexity:

Find ( ) : $O(\log n)$

Union ( ) : $O(1)$

* For every increase in depth of x by 1, size of the set containing x atleast DOUBLES.



So x goes into a set of sizes $1, \geq 2, \geq 4, \geq 8, \ldots$

doubling

Thus find() = $O(\log n)$

Pseudocode of Kruskal's Algorithm

$|V| = n$; $|E| = m$

Initialize:
$A = \phi$
e = 1st edge of L ← Sorted $\Rightarrow O(m \log m)$
Initialize the names of the connected components of $T = (V, A)$

Process:
while (NOT END of L):
let $e = (x, y)$
$X = Find(x)$
$Y = Find(y)$
if $(X \neq Y)$
$A \leftarrow A \cup \{e\}$
Union (X, Y)
$e = next(e, L)$

2m find() operations

(n-1) union() operations

Output:
$T = (V, A)$
// T is a MST of G

Total time Complexity:

time complexity of building L $= O(m \log m)$ (m = total # edges in the input graph)
+
2m Find()
+
(n-1) union()

n-Forest solution

$$T(n, m) = O(m \log m) + 2m \log n + (n-1)$$

$$T(n, m) = O(m \log n) + 2m + (n-1)n$$

$$= O(m \log n + m + n^2)$$

Name-Array implementation

$$T(n, m) = O(m \log n + n^2)$$

Name-Array vs In-Forest

	Name-Array	In-Forest
Find()	$O(1)$	$O(\log n)$
Union()	$O(n)$	$O(1)$
$T(n, m)$	$O(m \log m) + 2m\,O(1) + n\,O(n)$ $= O(m \log m + 2m + n^2)$ $= O(m \log m + n^2 + m)$	$O(m \log m) + 2m\,O(\log n) + (n-1)\,O(1)$ $= O(m \log m + 2m \log n + n - 1)$ $= O(m \log m + m \log n + n)$

If $m \log m$ is ignored, as a pre-processing cost,

Name-Array $\longrightarrow O(n^2 + m)$

In-Forest $\longrightarrow O(m \log n)$

Which is better?

Depends on the sparsity of the input graph,

Dense Graph $\Rightarrow$ $m = O(n^2)$ $\Rightarrow$ Name-array is preferred

Sparse Graph $\Rightarrow$ $m = O(n)$ $\Rightarrow$ in-forest is preferred

$\therefore$ Name-array based implementation $\rightarrow$ Dense Graphs
In-forest based implementation $\rightarrow$ Sparse Graphs