Lecture 97

Kruskal's algo | Disjoint Set | Union by rank & path compression.

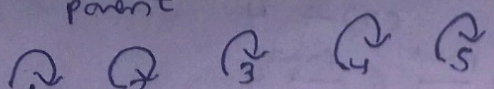
Disjoint Set (DS)

We use this Disjoint set to do Kruskal's algo.

1) Let say we have 2 nodes u and v, using DS we can check whether these 2 nodes, u and v belong to same component or Different Component of a graph.

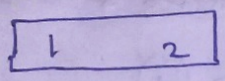
2) There are 2 important operations
   a) FindParent or FindSet()
   b) Union() or UnionSet()

3) Let say we have 5 disconnected Component where each node is its parent



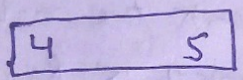
Find Parent (1) → 1
(2) → 2
⋮
(5) → 5

Union (1, 2)
we make single comp containing 1 and 2 where 1 is parent



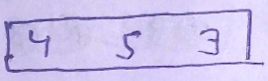
So now there is only one component for 1 & 2 instead of 2 seprate Components

Saveway
Union (4, 5)



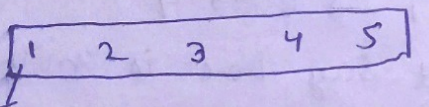
is a single Component now where 4 is parent

Union (3, 5)
now



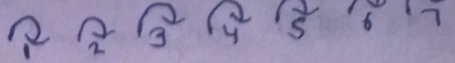
is a Component which is single where 4 is parent

Union (1, 3)



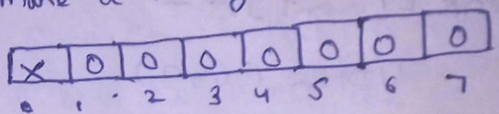
parent

Union by Rank & path Compression
Let say we have 7 disconnected Component



find Parent (1) → 1
(2) → 2
⋮
(7) → 7

To find union (1, 2) we make a Array of Rank

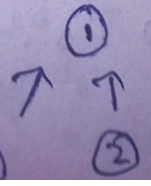


Initially Rank = 0 for all and we start from 1 so 0 = x

1) FindParent (1) and (2)
2) Check Rank of 1 and 2
if Rank(1) = Rank(2) we can attach anybody with anybody so
parent [2] = 1
Rank [1] ++

x, 1, 0 0 0 0 0 0

Union (2, 3)
1) Find Parent (2) → 1
and (3) → 3
2) Rank (1) = 1] → ≠
Rank (3) = 0
Rank (1) > Rank (3)
so parent [3] = 1

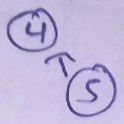


Union (4, 5)

1) Parent(4) → 4
   (5) → 5

2) Rank (4) = 0 = Rank(5)
   So attach anyone with anyone

3) Parent [5] = 4
   rank [4] ++



So when Rank is same we mark anyone as parent and do rank [Parent] ++

Union (6, 7)

1) Parent (6) → 6
   (7) → 7

2) Rank = same = 0 for 6, 7

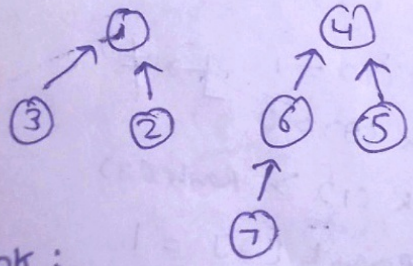
3) Parent [7] = 6
   Rank (6) ++

Union (5, 6)

1) Parent (5) → 4
   (6) → 6

2) Rank (4) = 1
   (6) = 1

3) Parent [6] = 4
   Rank [4] ++



Rank :

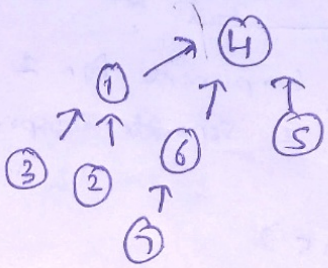
X	1	0	0	2	0	1	0
0	1	2	3	4	5	6	7

Union (3, 7)

1) Parent (3) → 1
   (7) → 4
   7 —parent→ 6 —parent→ 4
   So Parent (7) ≠ 6 = 4

2) Rank (1) = 1
   Rank (4) = 2

3) Rank (1) < Rank (4)
   So Parent [1] = 4



When Rank are not equal And Rank 1 < Rank 2 then Parent [Rank 1] = Rank 2

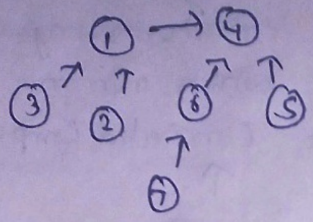
To find Parent (7)
we travelled a lot like
7 → 6 → 4
Let say there is another node (8) then we travel
8 → 7 → 6 → 4

So To optimise this time consuming thing, we use path compression

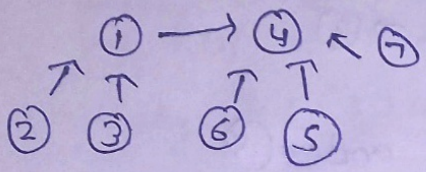
When we know

parent[7] = 6 and parent[6] = 4

So instead of



Do this



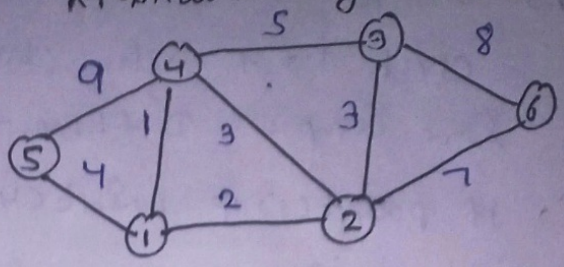
parent[7] = 4 now directly

So this is path compression logic

While doing union, we always try to merge a short Tree under a long tree so that depth do not increase whereas if we merge long tree under a short tree, length / depth of Tree ↑ which is not optimised way.

Above was logic for Disjoint Set now we see kruskal's algo.

kruskal's algo



To check whether 3, 4 lie in same component we check parent(3) and parent(4)

if both are =, they lie in same component

if both are ≠, they lie in Diffent Component

we need a

1) adj list ∝ (no need)
2) we need a linear DS Array to store u, v, weight in sorted order

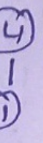
wt, u, v

Sorted		
wt	u	v
1	1	4
2	1	2
3	2	3
3	2	4
4	1	5
5	3	4
7	2	6
8	3	6
9	4	5

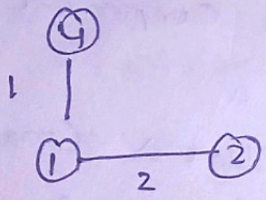
1 → 2, 2 ∝
1 → 4, 1 ∝
1 → 5, 4 ∝
4 → 5, 9 ∝
4 → 3, 5 ∝
2 → 4, 3 ∝
2 → 3, 3 ∝
2 → 6, 7 ∝
3 → 6, 8 ∝

we need to sort in order of weight, u and v can be in any order

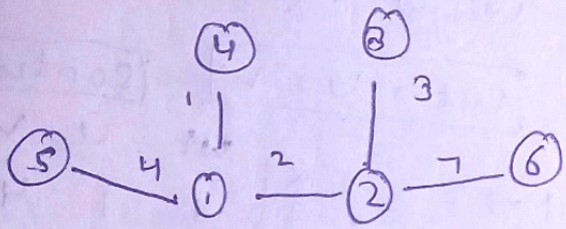
1) we get 1, 1, 4
check 1, 4 belong to
same comp or Different
if parent(1) ≠ parent(4)
do union
if parent(1) = Parent(4)
Do nothing



2) 2, 1, 2
check parent(1), parent(2)
merge as parent not equal



3) Do this for all element
from sorted list



This is our minimum spanning tree.

// TC
Sorting = $m \log m$, m = edges
Find parent or union takes
constant time

// TC = $O(m \log m)$
// SC = $O(n) + O(n)$
$\simeq O(n)$