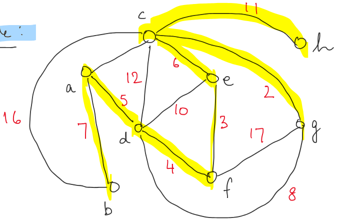
MST and Union-Find

Wednesday, 22 September 2021 2:04 PM

Kruskal's algorithm for finding a minimum weight spanning tree (MST) and the **Union-Find DS**

Example:

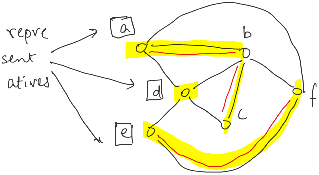


Component: A maximal, connected subgraph

i.e., given graph $G = (V, E)$ and edges $E' \subseteq E$, $V' \subseteq V$ is a component if:

(i) V' is connected in E';

(ii) there is no edge from $v \in V'$ to $w \in V \setminus V'$ in E' (V' is maximal)



$G = (V, E)$

— : E'

3 components

```
MST - Kruskal (G = (V,E), wts. w on edges) {
    T ← φ , i ← 1
    Sort edges so that w(e_1) ≤ w(e_2) ≤ ... ≤ w(e_m)
    while (|T| ≤ n-1) {
        Let e_i = (u, v)
```

```
            If $T \cup \{e_i\}$ is acyclic  → If u, v are in diff. components
                $T \leftarrow T \cup \{e_i\}$  → Put u, v in the same component
            $i \leftarrow i+1$
        }
    }
```

Claim: MST-Kruskal produces a minimum wt. spanning tree.

(prove yourself)

We will design a data structure that implements two operations:

1. Find (u) : returns which component vtx. u is in
2. Union (u, v) : If u, v are representatives of their components, then this merges the two components containing u & v.

```
MST_Kruskal ($G = (V, E)$, $w$) {
    Sort edges so that $w(e_1) \leq w(e_2) \leq \dots w(e_m)$
    $T \leftarrow \phi$, $i \leftarrow 1$
    for $i = 1 \dots m$
        Let $e_i = (u, v)$
        $f_u \leftarrow$ find (u), $f_v \leftarrow$ find (v)
        if ($f_u \neq f_v$) {                      ┐
            $T \leftarrow T \cup \{e_i\}$            │ $\leq n$ times
            Union ($f_u$, $f_v$)                   │
        }                                          ┘
        $i \leftarrow i+1$
    }
```

}

Option 1: use an array

A =

v_1	v_2	v_3	v_4	...	v_{n-1}	v_n

each vertex also stores pointer to it's representative / parent

Initially v.parent ← v

```
Find (u) { return u.parent }                     O(1)
Union (u, v)   // assume u.parent == u,  v.parent == v
{
    for i = 0 ... n
        if (A[i].parent == u)                    O(n)
            A[i].parent = v
}
```

Runtime algorithm : $O(m\log m + n^2)$

Currently : 2 issues with union:

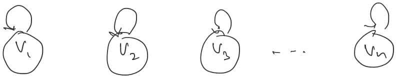
(i) it goes through entire array to find vertices in u's component

(soln: use pointers instead of an array to quickly find all vtxs. in u's component)

(ii) maintain size attribute for each component. In merging u & v, only change vertices of the smaller component.

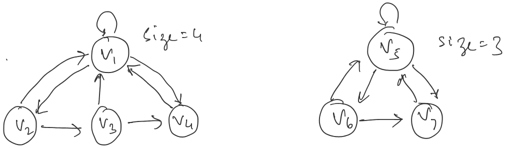
Will maintain : v. parent , v. size , v. firstchild , v. sibling, v. last child

Initially :

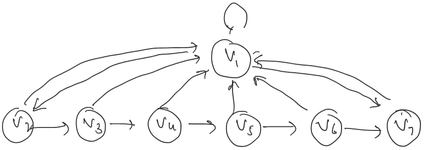


size = 1 for all, all other ptrs. are NULL

Later .



Union (v_1, v_5) :



Write down Code for Union (u,v) yourself

Find takes $O(1)$ time

Union in w.c. takes $O(n)$ time . But amortized ?

Claim : For any vertex v, parent can change at most $O(\log n)$ times

Proof : Initially size of v's component is 1.
Every time v's parent pointer is updated, it's component is merged with a larger component, hence size of v's component at least doubles ▨

Total run time over all Union operations is $O(n \log n)$

Hence runtime for algo is $O(m\log m + \underbrace{m + n\log n})$

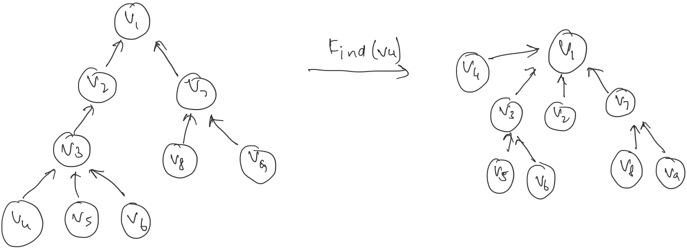
Suppose we give up on height 1 trees. Instead:

① Union just changes parent pointer of root node, thus takes $O(1)$ time.

② Instead of size, we will maintain <u>rank</u> of a component: roughly, an upper bound on height of a tree.

For union, component of smaller <u>rank</u> will be changed to point to component of larger rank.

③ Find will do path-compression.



```
Initially: v.parent ← v,   v.rank ← 0   ∀v
Union (u, v)  \\ assume u,v are roots {
    If (u.rank > v.rank)   v.parent ← u
    Else if (v.rank > u.rank)   u.parent ← v
    Else  v.parent ← u ;  u.rank ++   \\ if u.rank == v.rank
}

Find (v) {
    If (v.parent ≠ v)
```

```
        v.parent ← Find (v.parent)
    return v.parent
}
```

Analysis of Union & Find.

Properties : ① Find op: non-root vtxs. remain non-root vtxs.

Union op: non-roots remain non-root

⟹ once a vtx. is not a root, henceforth it is never a root

② Rank of a vtx. increases while it is a root node, then stays fixed once it is a non-root node

③ Suppose u is a parent of v. Then u.rank > v.rank

(proof by induction)

④ For any vertex u, it's rank. u.rank ⩾ height of subtree rooted at u.

(proof by induction, or from ③)

Claim 1: At any point during execution if v is root of a tree & has rank r, size of tree rooted at v is at least 2^r

Proof: By induction.

Base case: every vtx. is initially a component of size 1, rank 0, hence claim holds

Inductive step: Say next step is a find(u) operation. & v is root of tree containing u. Find$(\cdot)$ does not change either rank of root or nodes in the tree, hence the claim holds in this case.

Say next step is a Union(v, w) operation

Then size of tree rooted at $v \geq 2^{v.\text{rank}}$

" " " " " " $w \geq 2^{w.\text{rank}}$

Suppose $v.\text{rank} > w.\text{rank}$. Then v is root of new tree, rank remains unchanged, but size of tree increases, hence claim holds

Suppose $v.\text{rank} = w.\text{rank}$. Now rank of root of tree is $v.\text{rank}+1$. Size of the new tree $\geq 2^{v.\text{rank}} + 2^{w.\text{rank}} = 2^{v.\text{rank}+1}$. Hence claim continues to hold ∎

Claim 2: At any point during the algorithm, the number of nodes with rank $\geq r$ is at most $n/2^r$

Proof: Fix r. For vtx. v of rank $\geq r$, $\text{bucket}_r(v) = \{$ vtxs. that were descendents of v, when v just reached rank $r \}$. Let u be some other vtx. of rank $\geq r$, define $\text{bucket}_r(u)$ similarly. Note that $\text{bucket}_r(v) \geq 2^r$, $\text{bucket}_r(u) \geq 2^r$. To complete proof of claim, we will show that these buckets must be disjoint.

To see that $\text{bucket}_r(u) \cap \text{bucket}_r(v) = \phi$, say that v reached rank r before u. But v's parent in any step must have rank $> r$. So any vertex in v's bucket is either a descendent of v, or is a child of a node of rank higher than r. In either case, any vertex in v's bucket must be a descendant of a node of rank $\geq r$. But then none of these vertices could be in $\text{bucket}_r(u)$, since they would have to be in a tree where root has rank $r-1$. ▨

Define for $i \in \mathbb{Z}_+$, $\text{Tower}(i) = \underbrace{2^{2^{2^{\cdot^{\cdot^{2}}}}}}_{i}$

$\text{Tower}(0) = 2$, $\text{Tower}(1) = 4$, $\text{Tower}(2) = 16$,

$\text{Tower}(3) = 65536$, $\text{Tower}(4) = 2^{65536}$

In general, $\text{Tower}(i) = 2^{\text{Tower}(i-1)}$

The inverse operation is the iterated logarithm: $\log^* n$

(given n, how many times do I take the log to get 1?)

→ $\log^* n = \min\{i : \text{Tower}(i) \geq n\}$

$$\log^*(\log n) = \log^* n - 1$$

We will show that the amortized cost of k find operations is $O(k \log^* n)$

Note that union operation takes time $O(1)$ (worst case)

Hence run time of MST algo w/ union & find oprs. is $O(m \log m + n + m \log^* n)$

We will define different ranges for the ranks:

$\text{Range}(0) = [0, 2]$

$\text{Range}(1) = [3, 4]$ $\quad$ $\text{Range}(i) = [\text{Tower}(i-1)+1, \text{Tower}(i)]$

Range (3) = $[5, 16]$

Range (4) = $[17, 65536]$

Define Block (i) = $\{v :$ at end of algorithm, v.rank $\in$ Range(i)$\}$

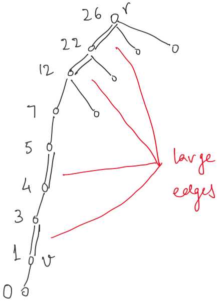
Now rank of any vertex $\leq \log n$, and hence for $i \geq \log^* n$, Block (i) is empty.

Since Range $(\log^* n)$ = $[\text{Tower}(\log^* n - 1) + 1, \ \text{Tower}(\log^* n)]$

$= [1 + \log n, \ n]$

but no vtx has rank in this interval, and hence Block (i) for $i \geq \log^* n$ must be empty.

Consider the operation find(v). Let r be the root of v's tree. Call the path from v to r the up-path from v.



Then find(v) takes time O (length of up-path from v).

We call edges either "small" or "large"

An edge (u, w) is a large edge if either:

① w = r, i.e., w is the last edge, OR

⑪ u & w are in different blocks, i.e., their ranks are in different ranges.

All other edges are small edges.

Q: In the up-path from v, how many large edges can I encounter?

A. $O(\log^* n)$, since every time I cross a large edge, I enter a higher range, & # ranges $\leq \log^* n$

Thus for k find operations, # of large edges encountered is $O(k \log^* n)$. Will show this bound holds for small edges also.

Fix $i \leq \log^* n$, fix a vertex v in Block (i). Thus,
$v.\text{rank} \in [\text{Tower}(i-1)+1, \text{Tower}(i)]$

Q. How many times over k find ops. will I encounter v as the lower vtx. of a small edge?

Note that: (i) v lower vtx $\Rightarrow$ v is non-root, so v's rank fixed

(ii) v's parent (say w) is in same block, hence $w.\text{rank} \in [\text{Tower}(i-1)+1, \text{Tower}(i)]$

(iii) Since (v, w) is a small edge, $w \neq$ root

Recall: ① each tree corresponds to a component, vtx. at root is representative of that component

② rank of a vertex in the data structure:

a. for any vertex v, $v.\text{rank} \leq \log n$

b. rank strictly increases as we go towards the root

Claim 1: At any point in the algorithm, if root of tree has rank r, size of tree $\geq 2^r$ (proved earlier)

Claim 2: At any point in the algorithm, # of vertices w/ rank $\geq r$ is at most $n/2^r$ (proved earlier)

③ From Claim 1: rank of any vertex $\leq \lceil \log n \rceil$

④ We divide ranks $0 \ldots \lceil \log n \rceil$ into blocks:

$\text{Tower}(-1) = 0$, $\text{Tower}(0) = 1$, $\text{Tower}(1) = 2$, $\text{Tower}(2) = 4$, ...

Tower (i) = $2^{\text{Tower}(i-1)}$

& Block (0) = $[0 \dots 0]$

Block (1) = $[1 \dots 2]$ Block (i) = $[\text{Tower}(i-1)+1 \dots \text{Tower}(i)]$

Block (2) = $[3 \dots 4]$

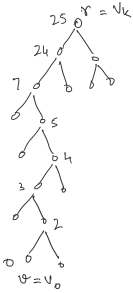
Block (3) = $[5 \dots 16]$

Block (4) = $[17 \dots 65536]$ ...

We informally say vtx $v \in$ Block (i) if v.rank $\in$ Block (i)
Clearly, from ③, there are only $1 + \lceil \log^* n \rceil$ blocks
(later blocks have no vertices)

⑤ Union opn. takes $O(1)$ time in worst case
We want to show that k find opns. take time
$O(k \log^* n)$.

Consider the opn Find (v). Say tree looks like:



Let r be root of v's tree. Define up-path as the path from v to r.

$p = (v = v_0, v_1, v_2, \dots, v_{k-1}, v_k = r)$

We'll say a vertex v_i on p appears small if v_i, v_{i+1} are in the same block

Conversely, v_i on p appears large if v_i, v_{i+1} are in different blocks

The root and it's children are always large.

Now cost of Find(v) is # of vertices in up-path.

# of small vertices

# of large vertices

Last time: On any up-path, # of large vertices is $O(\log^* n)$

Thus in k find opns., # of large vertices is $O(k \log^* n)$

Claim 3: In k find opns, the # of small vertices on the up-paths is $O(k \log^* n)$

To prove this for every vertex v, will obtain a bound on # of times this vtx. appears as a small vtx.

In 2 steps.

Step 1: Suppose we fix vtx. v in block i. Will show that vtx. v appears small at most Tower(i) times

Step 2: Will show that # vtxs. in block i is at most $\frac{n}{\text{Tower}(i)}$

Hence, # of times any vtx. from block i appears small is n

Hence, # of times any vtx. appears small is $n \log^* n$

Proof of Step 2: Block (i) contains ranks $[\text{Tower}(i-1)+1 \ldots \text{Tower}(i)]$

By Claim 2, # vtxs. of rank $\geq r$ is $\leq n/2^r$

So, # vtx. in block(i) $\leq \displaystyle\sum_{i=\text{Tower}(i-1)+1}^{\text{Tower}(i)} \frac{n}{2^i} \leq \frac{n}{2^{\text{Tower}(i-1)}} = \frac{n}{\text{Tower}(i)}$ ▨

(actually: $\leq \dfrac{n}{2^{\text{Tower}(i-1)+1}} \leq \dfrac{n}{\text{Tower}(i)}$

Proof of Step 1: Because v is a small vtx., it's own rank does not change. Secondly because it is small, it's parent is not the root. So during find operation, it will move up the tree & hence rank of it's parent strictly increases. But this can happen at most $|\text{Block}(i)| \leq \text{Tower}(i)$ times ▨