Minimum Spanning Tree (MST)

Input: (undirected connected) $G=(V,E,c)$

c_e is the cost of the edge e,

$c_e > 0$ and w.l.o.g all c_e's are distinct.

Output: a subset $T \subseteq E$ so that $G=\langle V,T\rangle$ is connected

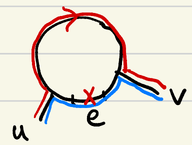
and the total cost $\sum_{e\in T} c_e$ is minimized.

Claim (4.16): T is a tree.

<u>Proof</u>: suppose T contains a cycle C,

then if we any edge $e\in C$, the graph

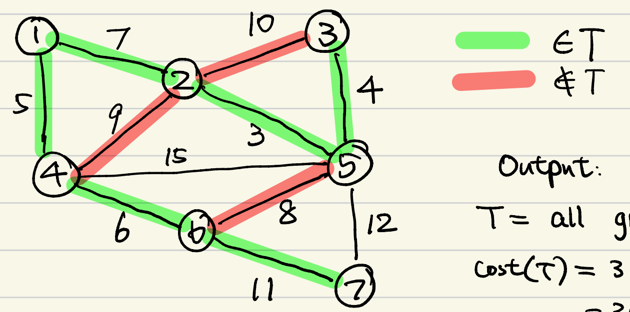
$(V, T-\{e\})$ is still connected.



Kruskal's algorithm:

1. Start with $T=\phi$.
2. Sort edges in E by cost in ascending order.
3. Iterate over all edges, add e to T as long as it does not create a cycle.

Example:



Output:

$T=$ all green edges

$\text{cost}(T) = 3+4+5+6+7+11$

$= 36$.

Correctness:

Theorem (4.18): Kruskal's algorithm produces an MST.
(page 146)

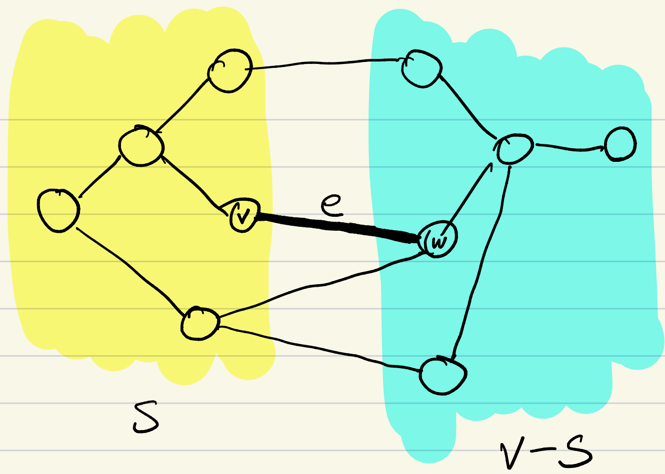
Lemma (4.17): Assume c_e's are distinct.

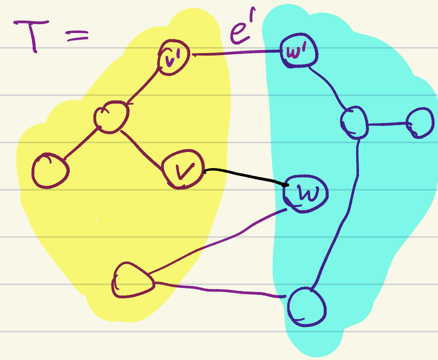
Fix $S\subseteq V$.

Let e be the minimum cost edge with one end in S and one end in $V-S$.

Then e must appear in every MST.

Proof.





Suppose $e=(v,w)$ is the cheapest edge between S and $V-S$.
Suppose there is an MST T s.t. $e \notin T$.

Because T is a spanning tree, there is a path P from v to w in T.
Since $v \in S$ and $w \notin S$, P must leave S at some point.
Let e' be the first edge in P that leaves S.
$e' = (v', w')$

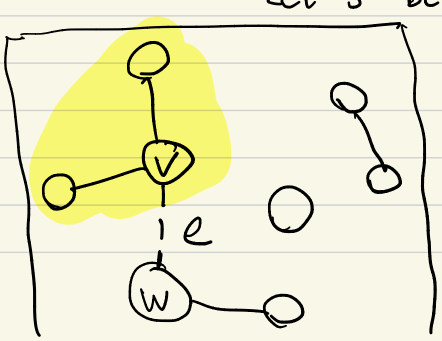
$T + \{e\} - \{e'\}$ is a spanning tree with smaller cost.

- the cost is smaller because $c_e < c_{e'}$.
- $T + \{e\}$ contains a unique cycle C and moreover $e' \in C$.

$\Rightarrow$ $T + \{e\} - \{e'\}$ is connected.
"$T + \{e\} - \{e\}$" has $(n-1)$ edges.

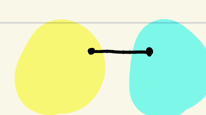
Proof (correctness of Kruskal):

Consider an edge $e=(v,w)$ added by Kruskal's algorithm.
Let S be v's connected component just before adding e.



- $w \notin S$ (o.w. adding e creates a cycle).
- no edge from S to $V-S$ has been considered (any such edge would have been added.)

Thus, e must be the cheapest edge that leaves S. so it is safe to include e.
Moreover, Kruskal's output must be connected.



Prim's algorithm.

1. Pick a root $s \in V$.
2. $S = \{s\}$. (All nodes in S are connected.)
3. Repeat the following until $S = V$:

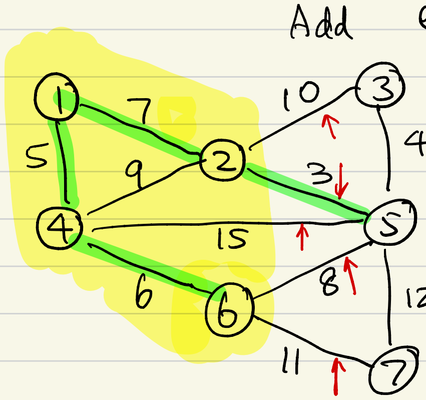
   Add v to S where $v \notin S$ minimizes the "attachment cost".
   (breaking ties arbitrarily)

   $$\min_{e=(u,v),\, u \in S} c_e.$$

   Add $e = (u,v)$ to T.

4. Return T.

Example:



Correctness:

- Every time the algorithm adds an edge e, e is the cheapest edge between S and $V-S$.
- T is a spanning tree at the end.

Runtime of Prim: $O(m \log n)$.

Similar to Dijkstra's algorithm.
We will use a min-heap H.

Overall $\begin{cases} O(n). & H.\min() \\ O(m) & H.\text{changekey}() \end{cases}$

each call takes $O(1)$ time
each call takes $O(\log n)$ time

Overall runtime $= O(n \cdot 1 + m \cdot \log n)$
$= O(m \log n)$.

Runtime of Kruskal:

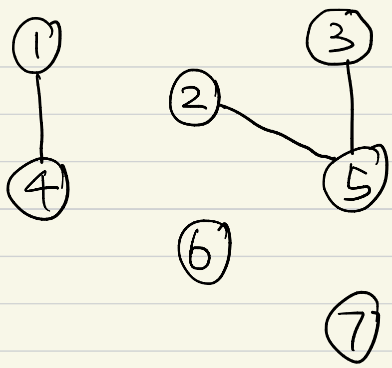
1. Sort all edges: $O(m \log m) = O(m \log n)$ $(m \le n^2)$
2. Check (in total $O(m)$ times) whether adding e would cause a cycle. How? ($O(n)$ via DFS)

   Can we do it faster?

The Union-Find data structure. (page 152)

- Init(): all element are in separate sets.
- Union(x, y): merge the set containing x with the set containing y.
- Find(x): return the "name" of the set that x is in.

We use this data structure to maintain the set of connected components (which are changing over time) in Kruskal's algorithm.



$\{\underline{1}, 4\}$ $\{\underline{2}, 3, 5\}$ $\{\underline{6}\}$ $\{\underline{7}\}$. representative element

When we consider any edge $e = (u, v)$

    If ($find(u) \neq find(v)$) then
        Add e to T
        union (u, v).
    End if.

Union-Find First Attempt.

arrays
- name[i] : name of the set that contains i.
- size[j] : size of the set with name j.

functions
- Init() : name[i] = i. size[i] = 1;
- Find(x) : return name[x];
- Union(x, y) :
    - x = name[x];
    - y = name[y];
    - If (size[x] < size[y]) swap(x, y);
      (so w.l.o.g size[x] $\geq$ size[y])
    - For every element i in set "y" :
        - name[i] = x;
    - End For
    - size[x] = size[x] + size[y];

Example: Before

	1	2	3	4	5	6	7 (index)
name :	1	2	2	1	2	6	7
size :	2	3				1	1

Union(4,5) $x = 4$. $y = 5$ $\Rightarrow$ $x = 1$ $y = 2$.

$x = 2$ $y = 1$ (swapped so that size[x] $\geq$ size[y])

For $i \in \{1, 4\}$.
    name[i] = 2
End for
size[2] = size[2] + size[1].

After.

	1	2	3	4	5	6	7 (index)
name	2	2	2	2	2	6	7
size		5				1	1

We always merge the smaller set into the bigger one!

Runtime: Init: $O(n)$. $\times O(1)$ Find : $O(1)$ $\times O(m)$

Claim: The first k union operations run in total time $O(k \log k)$. $k = n$

$\Longrightarrow$ Kruskal runs in time : $O(\underbrace{m \log n}_{\text{sorting}} + \underbrace{m}_{\text{find}} + \underbrace{n \log n}_{\text{union}}) = O(m \log n)$

Proof sketch:

After $k>1$ union operations, the largest set has size $O(k)$.

How many times can name[i] change?

Once per union(A, B) if $i \in B$ and $|B| \le |A|$.

At most $O(\log k)$ times because the size of the set name[i] at least doubles each time.

At most $2k$ name[i] are changed.

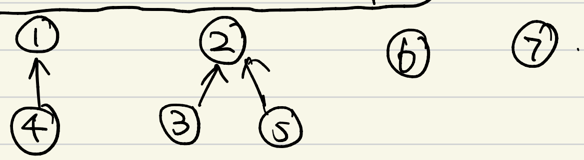
$\Downarrow$

$O(k \log k)$.

Not very ideal because sometimes Union(x,y) can take $\Omega(n)$ time.

Example: name = $\underbrace{1\,1 \cdots 1\,1}_{n/2}\ \underbrace{2\,2 \cdots 2}_{n/2}$

Union-Find Second Attempt



index	1	2	3	4	5	6	7	(-1 if no parent)
parent	-1	-1	2	1	2	-1	-1	

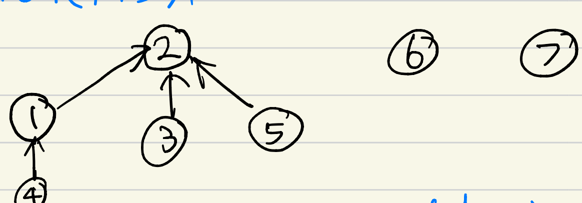
Init() : parent[i] = -1 $\forall i$.

Find(x) : return the root of x.

Find(x) :
while (parent[x] ≠ -1)
x = parent[x]
end while
return x

Union(x, y) : x = Find(x), y = Find(y).
merge the smaller set into the bigger one.
If size[x] ≥ size[y]
parent[y] = x ; size[x] = size[x] + size[y]
Else
parent[x] = y . size ⋯ -

Example. Union(4, 5).



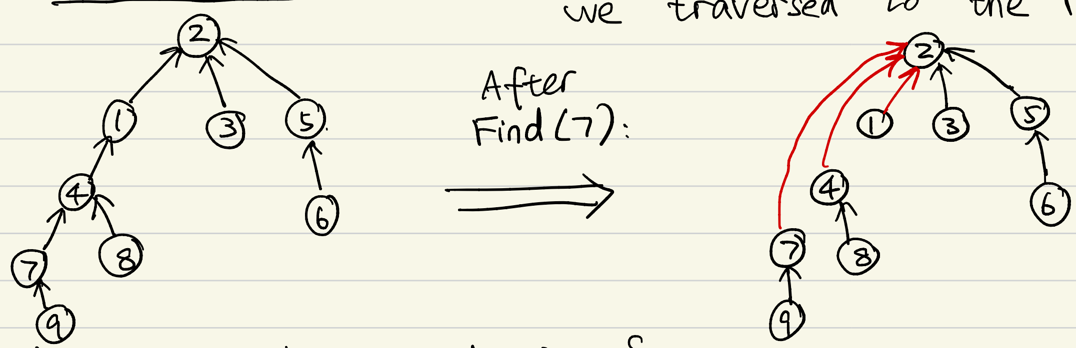
Runtime:
Init() : $O(n)$
Find() : $O(\log n)$
the depth of any tree is at most $O(\log n)$
Union() : $O(1)$ after x=find(x) and y=find(y).

Runtime of Kruskal under this Union-Find implementation $= O(\underbrace{m\log n}_{\text{sorting}} + \underbrace{m \cdot \log n}_{\text{find}} + \underbrace{n}_{\text{union}})$

Union-find Third Attempt (the real version)

Building on the second attempt, we add one more optimization.

Path compression: After find(x), point all the elements we traversed to the root.



Implementation:

```
int Find(x). {
    If (x == parent[x]) return x;
    root = Find(parent[x]);   ← recursion.
    parent[x] = root;
    return root
}
```

find(7) → find(4) → find(1) → find(2) = 2

parent[7]=2 ← parent[4]=2 ← parent[1]=2 ←

Upshot: Next time we call find(7) before any union operation takes $O(1)$ time.

Runtime: First k Union/Find opearations

A "simple" proof (available on Union-Find Wikipedia page) shows that these k operations runs in $O(k \cdot \log^* n)$. [Hopcroft and Ullman '73]

$\log^* n$: iterated logarithm: # of times log needs to be applied for n to become 1.

$$\log_2^*\left(2^{2^{2^{2^{2}}}}\right) = 5$$

$$\approx 10^{19728}$$

$O(m \cdot \alpha(n))$ [Tarjan '75] $\qquad \Theta(m \cdot \alpha(n))$ [Fredman and Saks '89]

inverse Ackermann function (grows even slower than $\log^* n$).