Minimum Spanning Tree

- The Cycle Property (Red Rule)
  - Reverse Delete Algorithm for MST
- Kruskal's Algorithm for MST
- Runtime and Implementation
  - Disjoint Sets Data Structure

IMDAD ULLAH KHAN

Kruskal's Algorithm

Algorithm Kruskal's Algorithm, $G = (V, E, w)$

Sort edges in increasing order of weights ▷ let $e_1, e_2, \ldots, e_m$ be the sorted order

$F \leftarrow \emptyset$ ▷ Begin with a forest with no edges

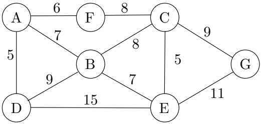
for $i = 1$ to m **do**

  if $F \cup e_i$ does not contain a cycle **then**

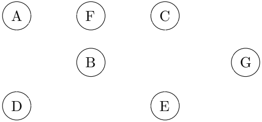
    $F \leftarrow F \cup \{e_i\}$

return F

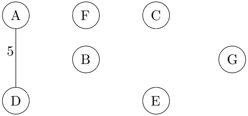
Kruskal's Algorithm: Example



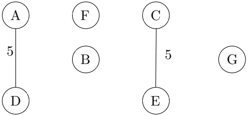
$G = (V, E, w)$



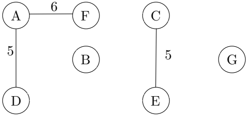
Initially each vertex is a tree



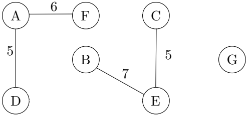
(A, D) is picked for merging



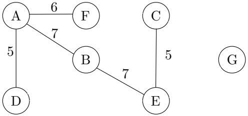
(C,E) is picked for merging



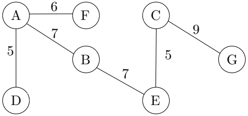
(A, F) is picked for merging



(B, E) is picked for merging



(A, B) is picked for merging



(C, G) is merged skipping $(F, C), (B, C), \ldots$

Kruskal's Algorithm: Runtime of Naive Implementation

Algorithm Kruskal's Algorithm, $G = (V, E, w)$

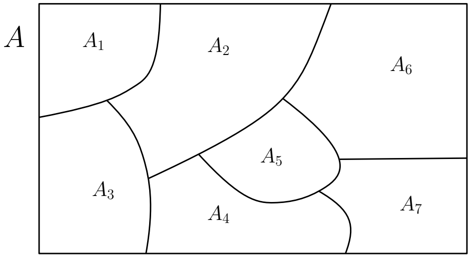
Sort edges in increasing order of weights ▷ $e_1, e_2, \ldots, e_m$ is sorted order
$F \leftarrow \emptyset$
for $i = 1$ to m **do**
  if $F \cup e_i$ does not contain a cycle **then**
    $F \leftarrow F \cup \{e_i\}$
return F

- Sorting takes $O(m \log m) = O(m \log n)$ time
- Detecting cycles in $F \cup \{e_i\}$ can be done by DFS
- $F \cup \{e_i\}$ has at most n vertices and $n - 2$ edges
- Total runtime $O(m \log n) + O(m \cdot (n + n))$
- Can do better using integer sorting or if input is already sorted
- Repeated cycle detection is bottleneck ▷ the 2nd term

Set Partition

Given a set A, $\mathcal{P} = \{A_1, \ldots, A_k\}$ is a partition of A if

- $A_i \subset A$ for $1 \leq i \leq k$
- $A_i \cap A_j = \emptyset$ for $1 \leq i \neq j \leq k$
- $A_1 \cup A_2 \cup \ldots \cup A_k = A$



UNION-FIND data structure

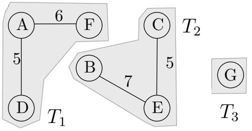
- Also known as disjoint sets data structure
- Maintains a partition of a set A
- Supports the following operations
  1. $\text{MAKESET}(x)$: creates a subset of size 1
  2. $\text{FIND}(x)$: returns id of the set containing x
  3. $\text{UNION}(x, y)$: union(merge) the sets containing x and y

---

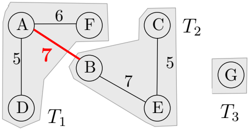
- F induces a partition of V
- Store F as the above data structure
- Every tree in F is a subset of V
- Edge (u, v) creates a cycle if u and v are in the same tree
- Edge (u, v) creates a cycle $\leftrightarrow$ $\text{FIND}(u) = \text{FIND}(v)$
- Pick edge (u, v) $\leftrightarrow$ $\text{UNION}\left(\text{FIND}(u), \text{FIND}(v)\right)$

UNION-FIND data structure

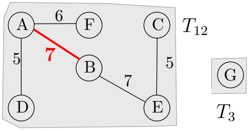
- F induces a partition of V
- Store F as the above data structure
- Every tree in F is a subset of V
- Edge (u, v) creates a cycle if u and v are in the same tree
- Edge (u, v) creates a cycle $\leftrightarrow$ $\textsc{find}(u) = \textsc{find}(v)$
- Pick edge (u, v) $\leftrightarrow$ $\textsc{union}\left(\textsc{find}(u), \textsc{find}(v)\right)$



Forest with 3 trees



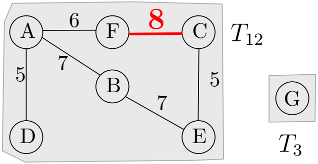
Pick $(A, B) \to$ Merge T_1 and T_2



T_1 and T_2 merged into T_{12}

UNION-FIND data structure

- F induces a partition of V
- Store F as the above data structure
- Every tree in F is a subset of V
- Edge (u, v) creates a cycle if u and v are in the same tree
- Edge (u, v) creates a cycle $\leftrightarrow$ $\text{FIND}(u) = \text{FIND}(v)$
- Pick edge (u, v) $\leftrightarrow$ $\text{UNION}\left(\text{FIND}(u), \text{FIND}(v)\right)$



Adding edge (F, C) creates a cycle
$\text{FIND}(F) = \text{FIND}(C)$

Kruskal's Algorithm with UNION-FIND

Algorithm Kruskal's Algorithm with UNION-FIND

```
for v ∈ V do
    MAKESET(v)
Sort edges in increasing order of weights
F ← ∅
for i = 1 to m do                e_i = (u, v)
    if FIND(u) ≠ FIND(v) then
        F ← F ∪ {e_i}
        UNION(u, v)
return F
```

$$\text{Runtime} \quad : \quad \sum \begin{cases} O(n) & \textsc{Makeset} \\ O(n) & \textsc{Union} \\ O(m) & \textsc{Find} \end{cases}$$

Union-Find Data Structure: Implementation

- Maintains a partition of a set A
- Supports the following operations
- MAKESET(x): creates a subset of size 1
- FIND(x): returns id of the set containing x
- UNION(x, y): union(merge) the sets containing x and y

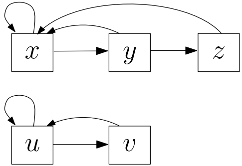
- Store each subset as a linked list
- Each node of the list has a pointer to the first
- The first node (an element of the subset) is the **rep** of the list
- **rep** of a list serves as an **id** of the subset

Node
class/struct

```
int element
Node* next
Node* rep
```

$A_1 = \{x, y, z\}$
$rep(A_1) = x$

$A_2 = \{u, v\}$
$rep(A_2) = u$

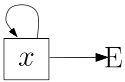


Union-Find Data Structure: Implementation

- MakeSet(u):
- Make a new list node rep-pointer to itself
- Store pointer to node in $P[u]$ (array indexed by A)
- Runtime $O(1)$

---

function MAKESET(u)
  $ptr \leftarrow$ NEW($Node$)
  $ptr \cdot element \leftarrow u$
  $ptr \cdot next \leftarrow null$
  $ptr \cdot rep \leftarrow ptr$
  $P[u] \leftarrow ptr$

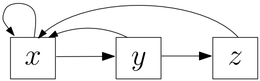


UNION-FIND Data Structure: Implementation

- FIND(u):
- Get pointer from $P[u]$
- Return vertex name at rep-pointer of node at $P[u]$
- Runtime $O(1)$

---

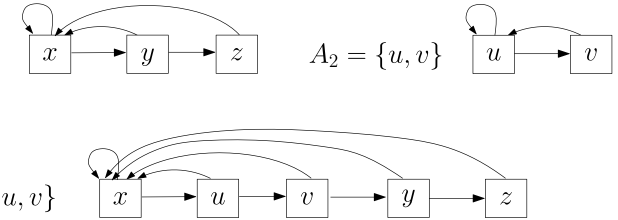
function FIND(u)
 $ptr \leftarrow P[u]$
 $rep \leftarrow ptr \cdot rep$
 return $rep \cdot element$



Union-Find Data Structure: Implementation

- UNION(u, v):
- Get pointers from $P[u]$ and $P[v]$
- Add $List_u$ to $List_v$ (say starting from second node)
- update rep-pointers at all nodes in $List_u$
- Runtime $O(1) + O(|List_u|)$

$A_1 = \{x, y, z\}$



UNION-FIND Data Structure: Implementation

Algorithm Kruskal's Algorithm with UNION-FIND

for $v \in V$ **do**
  MAKESET(v)
Sort edges in increasing order of weights
$F \leftarrow \emptyset$
for $i = 1$ to m **do** $e_i = (u, v)$
  if FIND(u) $\neq$ FIND(v) **then**
    $F \leftarrow F \cup \{e_i\}$
    UNION(u, v)
return F

$$\text{Runtime} \quad : \quad \sum \begin{cases} O(n) & \text{MAKESET} \\ O(n) & \text{UNION} \\ O(m) & \text{FIND} \end{cases}$$

Worst case: A list length could be $O(n)$

Union by rank

- In the first node save length of the list
- Called rank of the set (cardinality)
- For UNION(u, v) insert smaller rank set into bigger
- potentially fewer rep-updates common sense
- A little more careful analysis lead to see the power of this simple rule
- Every time a $rep(u)$ is updated its new list is at least doubled
- Max number of *rep* updates per element (vertex): $O(\log n)$
- Total rep updates for V is $O(n \log n)$
- So total runtime of all UNION$(\cdot, \cdot)$ is $O(n) + n \log n$