



# Algorithms and Data Structures

## Lecture 18 Kruskal's Algorithm and Disjoint Sets

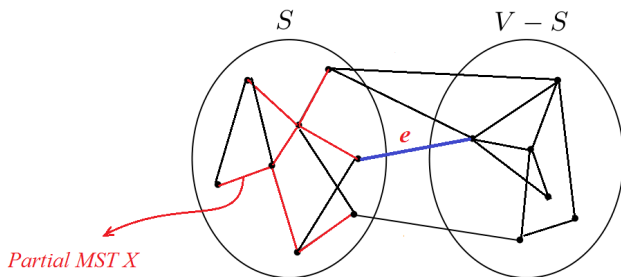
Jiamou Liu  
The University of Auckland



# Cut Property Revisited

Let  $G = (V, E, w)$  be a weighted graph.

- Suppose we have constructed a partial MST  $X$  on a subset  $S \subseteq V$ .
- Let  $e$  be the lightest edge across the partition between  $S$  and  $V - S$ .
- Then  $X \cup \{e\}$  is also a partial MST.

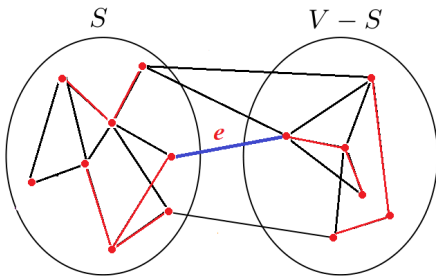


## Cut Property (Version 2.0)

Let  $G = (V, E, w)$  be a weighted graph.

- We say a **partial MSF** of  $G$  is a subset of edges that could lead to an MST.
- Suppose we have constructed a partial MSF  $X$ .
- Let  $e$  be the lightest edge across any two trees in this partial MSF.
- Then  $X \cup \{e\}$  is also a partial MSF.

Instead of Partial  
MST, we could allow  
*partial minimal  
spanning forest*



# Kruskal's Algorithm

Version 2.0 of the cut property allows us to conceptually simplify Prim's algorithm:

- We do not need to make a **traversal**.
- In other words, we do not need to keep the **known region** connected.
- Eventually, all the disconnected parts will link together to form an MST.

## Idea

Add edges into the MSF one-by-one:

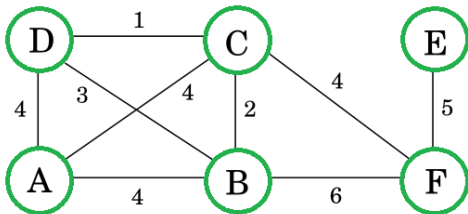
- In increasing order of the weights
- Make sure no cycle is created

Invented by Joseph Kruskal in 1956.

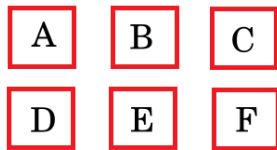
## Idea

Add edges into the MSF one-by-one:

- In increasing order of the weights
- Make sure no cycle is created



## Disjoint Sets

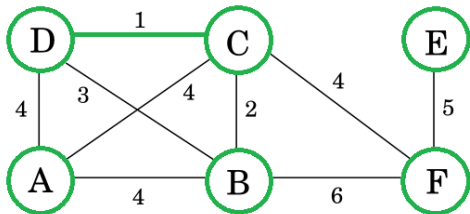


Invented by Joseph Kruskal in 1956.

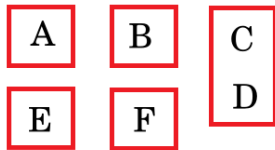
## Idea

Add edges into the MSF one-by-one:

- In increasing order of the weights
- Make sure no cycle is created



Disjoint Sets

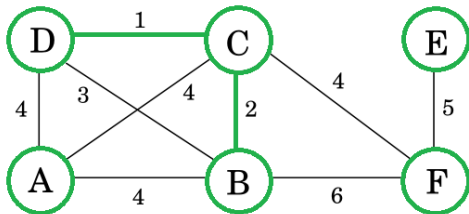


Invented by Joseph Kruskal in 1956.

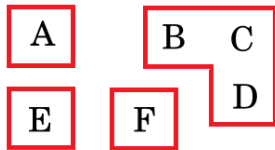
## Idea

Add edges into the MSF one-by-one:

- In increasing order of the weights
- Make sure no cycle is created



Disjoint Sets

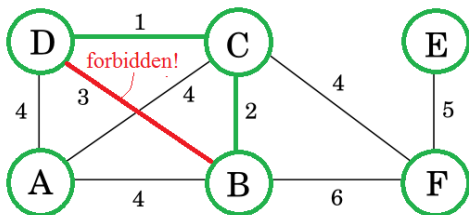


Invented by Joseph Kruskal in 1956.

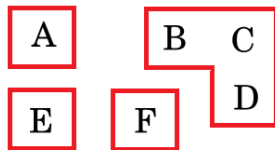
## Idea

Add edges into the MSF one-by-one:

- In increasing order of the weights
- Make sure no cycle is created



Disjoint Sets

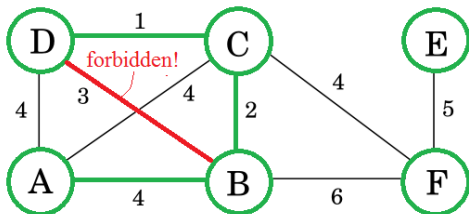


Invented by Joseph Kruskal in 1956.

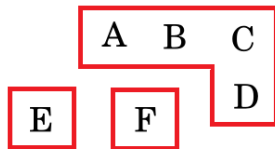
## Idea

Add edges into the MSF one-by-one:

- In increasing order of the weights
- Make sure no cycle is created



Disjoint Sets

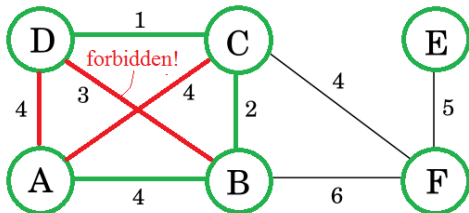


Invented by Joseph Kruskal in 1956.

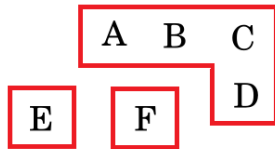
## Idea

Add edges into the MSF one-by-one:

- In increasing order of the weights
- Make sure no cycle is created



Disjoint Sets

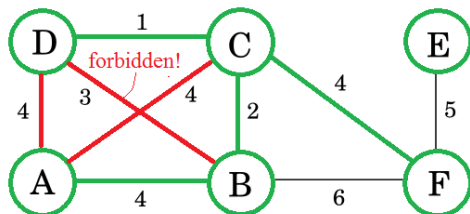


Invented by Joseph Kruskal in 1956.

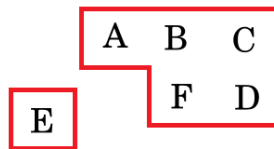
## Idea

Add edges into the MSF one-by-one:

- In increasing order of the weights
- Make sure no cycle is created



Disjoint Sets

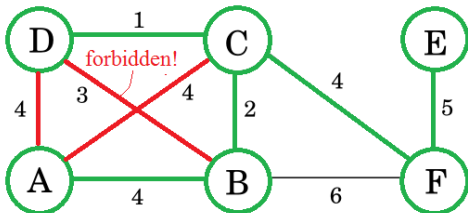


Invented by Joseph Kruskal in 1956.

## Idea

Add edges into the MSF one-by-one:

- In increasing order of the weights
- Make sure no cycle is created



Disjoint Sets

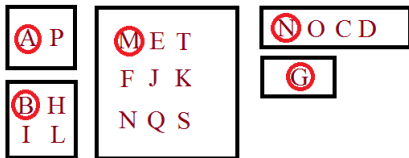
A	B	C
E	F	D

Kruskal's algorithm keeps a data structure for identifying forbidden edges.

## Definition

A **disjoint-sets data structure** maintains a collection of disjoint sets such that each set has a unique **representative element** and supports the following operations:

- **MakeSet( $u$ )**: Make a new set containing element  $u$ .
- **Union( $u, v$ )**: Merge the sets containing  $u$  and  $v$ .
- **Find( $u$ )**: Return the representative element of the set that contains  $u$



### **MST\_Kruskal**( $V, E, w$ )

1. Sort edges in  $E$  in increasing weights, store in a list **SortedEdges**
2. Initialize a **disjoint-sets** with each node a separate set
3. Initialize an empty set **X**
4. **for** each  $\{u, v\} \in \text{SortedEdges}$  **do**
5.     **if**  $\text{find}(u) \neq \text{find}(v)$  **then**
6.          $X \leftarrow X \cup \{\{u, v\}\}$
7.          $\text{union}(u, v)$
8. **return** **X**

The set  $X$  contains all edges in an MST.

# Kruskal's Algorithm: Complexity

## Running Time Analysis

The running time of Kruskal's algorithm depends on

- The complexity of the sorting algorithm
- The complexity of union-find operations

# Kruskal's Algorithm: Complexity

## Running Time Analysis

The running time of Kruskal's algorithm depends on

- The complexity of the sorting algorithm
- The complexity of union-find operations

Define:

- $T_{\text{sort}}(x)$  = time to sort  $x$  elements
- $T_{\text{find}}(x)$  = time to find an element
- $T_{\text{union}}(x)$  = time to take the *union* of two sets

# Kruskal's Algorithm: Complexity

## Running Time Analysis

The running time of Kruskal's algorithm depends on

- The complexity of the sorting algorithm
- The complexity of union-find operations

Define:

- $T_{\text{sort}}(x)$  = time to sort  $x$  elements
- $T_{\text{find}}(x)$  = time to find an element
- $T_{\text{union}}(x)$  = time to take the *union* of two sets

Running time for **MST\_Kruskal**:  $O(T_{\text{sort}}(m) + T_{\text{find}}(x)m + T_{\text{union}}(x)n)$

# Disjoint-Sets: Implementation 1

## Disjoint-Sets: Lists

**K** L O P Y

**A** X R Q

**U** V B

**C** D H M N W Z

Each set is presented by an array.

The **representative** is the first element

- Union:
- Find:

Therefore the running time of Kruskal's algorithm: .

# Disjoint-Sets: Implementation 1

## Disjoint-Sets: Lists

**K** L O P Y

**A** X R Q

**U** V B

**C** D H M N W Z

Each set is presented by an array.

The **representative** is the first element

- Union:  $O(1)$
- Find: Need to go through the list  $O(n)$

Therefore the running time of Kruskal's algorithm:

# Disjoint-Sets: Implementation 1

## Disjoint-Sets: Lists

**K** L O P Y

**A** X R Q

**U** V B

**C** D H M N W Z

Each set is presented by an array.

The **representative** is the first element

- Union:  $O(1)$
- Find: Need to go through the list  $O(n)$

Therefore the running time of Kruskal's algorithm:  $T_{\text{sort}}(m) + O(mn)$ .

# Disjoint-Sets: Implementation 2

## Disjoint-Sets: Trees

- Each set is represented by a **tree**. The **representative** is the root.
- Each node is associated with a **rank**, i.e., the height of its subtree.
- **Union**: link two trees; point the root with lower rank to the root with higher rank.
- **Find**: follow parent pointers to find the root.

$\text{makeset}(A), \text{makeset}(B), \dots, \text{makeset}(G):$

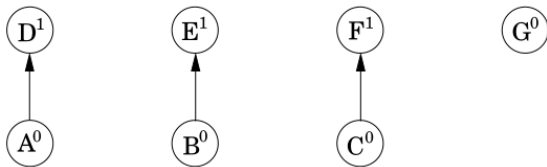


# Disjoint-Sets: Implementation 2

## Disjoint-Sets: Trees

- Each set is represented by a **tree**. The **representative** is the root.
- Each node is associated with a **rank**, i.e., the height of its subtree.
- Union**: link two trees; point the root with lower rank to the root with higher rank.
- Find**: follow parent pointers to find the root.

$\text{union}(A, D), \text{union}(B, E), \text{union}(C, F):$

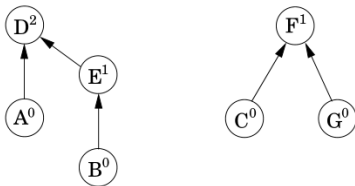


# Disjoint-Sets: Implementation 2

## Disjoint-Sets: Trees

- Each set is represented by a **tree**. The **representative** is the root.
- Each node is associated with a **rank**, i.e., the height of its subtree.
- Union**: link two trees; point the root with lower rank to the root with higher rank.
- Find**: follow parent pointers to find the root.

$\text{union}(C, G), \text{union}(E, A):$

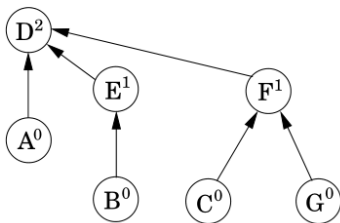


# Disjoint-Sets: Implementation 2

## Disjoint-Sets: Trees

- Each set is represented by a **tree**. The **representative** is the root.
- Each node is associated with a **rank**, i.e., the height of its subtree.
- Union**: link two trees; point the root with lower rank to the root with higher rank.
- Find**: follow parent pointers to find the root.

$\text{union}(B, G):$



- **Union**: link two trees; point the root with lower rank to the root with higher rank.  
Running time:  $O(1)$
- **Find**: follow parent pointers to find the root.  
Running time: Depend on the **height (rank)** of the tree.

- **Fact 1.** A node with rank  $k$  must have at least  $2^k$  nodes in its subtree.

- **Fact 1.** A node with rank  $k$  must have at least  $2^k$  nodes in its subtree.

Why? Can be proved by induction on  $k$ .

- **Fact 1.** A node with rank  $k$  must have at least  $2^k$  nodes in its subtree.

Why? Can be proved by induction on  $k$ .

- **Fact 2.** Let  $k$  be the largest rank. There can be at most  $n/2^k$  nodes of rank  $k$ .

- **Fact 1.** A node with rank  $k$  must have at least  $2^k$  nodes in its subtree.

Why? Can be proved by induction on  $k$ .

- **Fact 2.** Let  $k$  be the largest rank. There can be at most  $n/2^k$  nodes of rank  $k$ .

⇒ the largest rank  $k$  is at most  $\log n$ .

⇒  $\text{Find}(u)$  takes time  $O(\log n)$ .

- **Fact 1.** A node with rank  $k$  must have at least  $2^k$  nodes in its subtree.

Why? Can be proved by induction on  $k$ .

- **Fact 2.** Let  $k$  be the largest rank. There can be at most  $n/2^k$  nodes of rank  $k$ .

⇒ the largest rank  $k$  is at most  $\log n$ .

⇒  $\text{Find}(u)$  takes time  $O(\log n)$ .

Therefore Kruskal's algorithm takes time  $T_{\text{sort}}(m) + O(m \log n)$ .

## Different Disjoint-Sets

Kruskal's algorithm has different running time for different disjoint-set implementations:

- Arrays:
- Trees:

## Different Disjoint-Sets

Kruskal's algorithm has different running time for different disjoint-set implementations:

- **Arrays:**  $T_{\text{sort}}(m) + O(mn)$
- **Trees:**  $T_{\text{sort}}(m) + O(m \log n)$
- **Trees with Path Compression:**  $T_{\text{sort}}(m) + O(m \log^* n)$ ,  
where  $\log^* n$  is  $O(\underbrace{\log \log \dots \log n}_k)$  for any  $k > 0$ .

(typically called the **iterated logarithmic function**.)

- Cut property Version 2: Extending to forests
- Kruskal's algorithm
  - Correctness
  - Complexity
- Disjoint-set data structure
  - Implementation 1: List
  - Implementation 2: Tree

