CSI - 3105 Design & Analysis of Algorithms
Course 11

Jean-Lou De Carufel

Fall 2019

Minimum Spanning Tree

We are given a graph $G = (V, E)$ that is undirected and connected. Each edge $\{u, v\} \in E$ has a weight $wt(u, v)$.

We want to compute a subgraph G' of G such that

- The vertex set of G' is V,
- G' is connected,
- and $weight(G')$ is minimum, where

$$weight(G') = \text{sum of weights of edges in } G'.$$

Minimum Spanning Tree

We are given a graph $G = (V, E)$ that is undirected and connected. Each edge $\{u, v\} \in E$ has a weight $wt(u, v)$.

We want to compute a subgraph G' of G such that

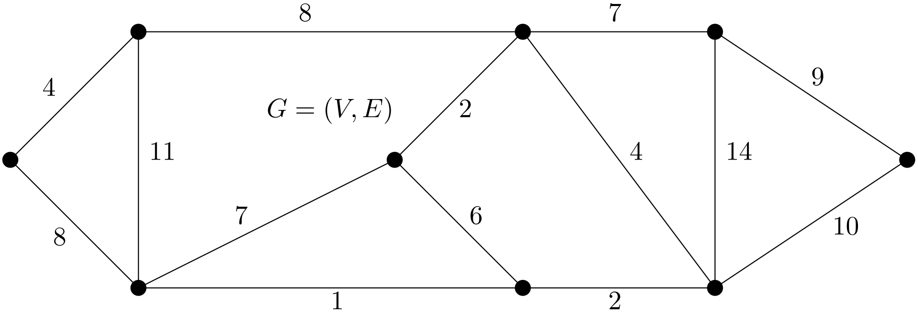
- The vertex set of G' is V,
- G' is connected,
- and $weight(G')$ is minimum, where

$$weight(G') = \text{sum of weights of edges in } G'.$$

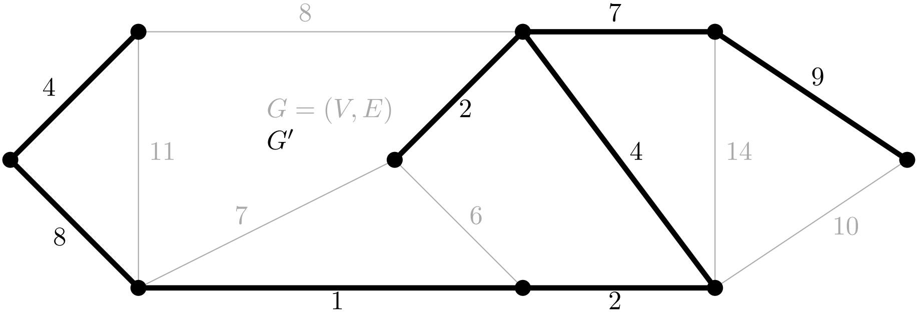
We can prove that G' must be a tree (connected and no cycles). Do you see why?

G' is called a *Minimum Spanning Tree of G* (MST of G).

Example:



Example:



- The vertex set of G' is V,
- G' is connected,
- and $weight(G')$ is minimum, where

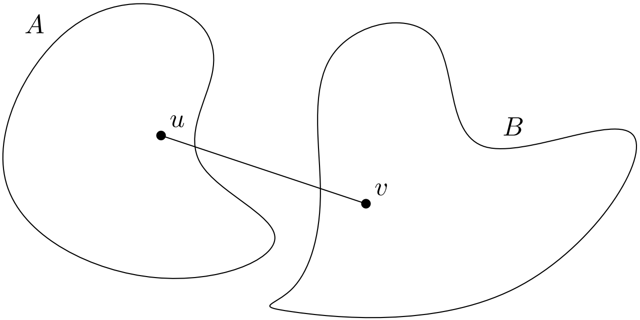
$$weight(G') = \text{sum of weights of edges in } G'.$$

Fundamental Lemma

Lemma

Let $G = (V, E)$ be an undirected and connected graph, where each edge $\{u, v\} \in E$ has a weight $wt(u, v)$.

Split V into A and B. Let $\{u, v\} \in E$ be a shortest edge connecting A and B. Then there is an MST of G that contains $\{u, v\}$.



PROOF:

From the previous lemma, any algorithm that follows this greedy scheme is guaranteed to work:

- $X = \{\ \}$ //edges picked so far
- Repeat until $|X| = |V| - 1$
  - Pick a set S such that X has no edge between V and $V \setminus S$.
  - Let $e \in E$ be a minimum-weight edge between V and $V \setminus S$.
  - $X = X \cup \{e\}$

About the Union-Find Data Structure

$$\Bigg($$

Before presenting a first algorithm to compute an MST, we first open a parenthesis and study a data structure called *Union-find*.

Given n sets, each of size one,

$$A_1 = \{1\}, \quad A_2 = \{2\}, \qquad \cdots \qquad A_n = \{n\},$$

process a sequence of operations, where each operation is one of

Union(A, B, C):

Set $C = A \cup B$

$A = \{\ \}$

$B = \{\ \}$

Find(x):

Return the name of the set that contains x.

The sequence consists of

$n-1$ **Union** operations

m **Find** operations

which can be done in any arbitrary order.

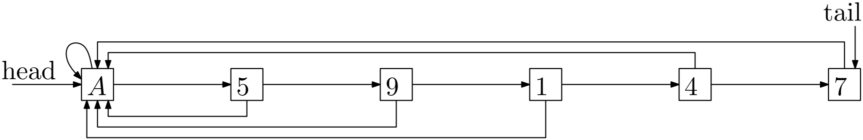
We are interested in the total time to process any such sequence.

Store each set in a list:

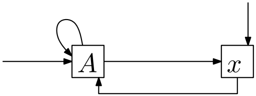
- the list has a pointer to the head and a pointer to the tail
- the first node stores the name of the set
- each other node stores one element of the set
- each node u stores two pointers:

  next(u) the next node in the list
  back(u) first node in the list

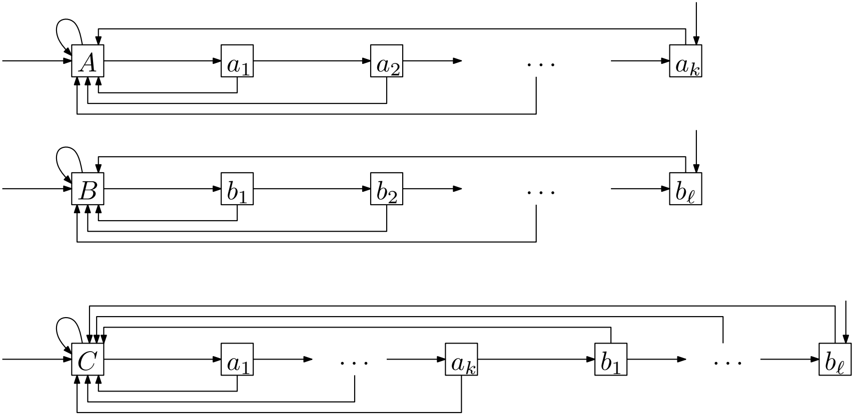
$A = \{1, 4, 5, 7, 9\}$



Start: for each set $A = \{x\}$:

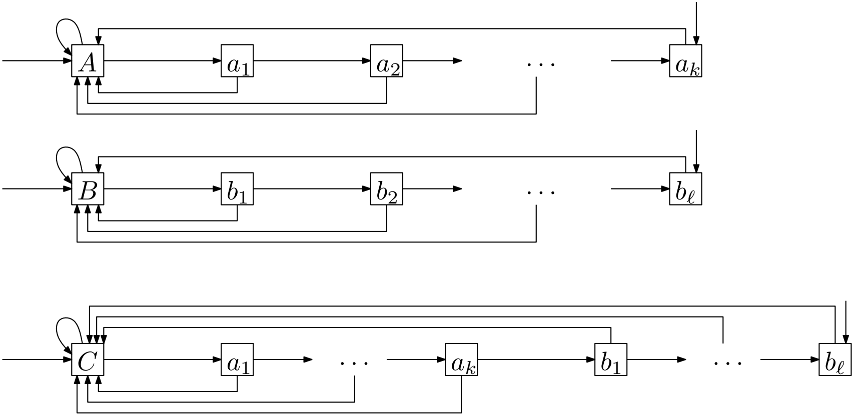


Union(A, B, C):



Append the list B at the end of the list A, do some pointer arithmetic, change the name in the head of the new list from A to C.

Union(A, B, C):



Time $= O(\ell) = O(\text{size of } B)$

Find(x): follow the back pointer from the node storing x to the head of the list and return the name stored at the head.

Time $= O(1)$

Example:

Union	Time
$\{2\}, \{1\}$	1
$\{3\}, \{2, 1\}$	2
$\{4\}, \{3, 2, 1\}$	3
$\vdots$	$\vdots$
$\{n\}, \{n-1, n-2, ..., 2, 1\}$	$n-1$

Example:

Union	Time
$\{2\}, \{1\}$	1
$\{3\}, \{2, 1\}$	2
$\{4\}, \{3, 2, 1\}$	3
$\vdots$	$\vdots$
$\{n\}, \{n-1, n-2, ..., 2, 1\}$	$n-1$

Total time $= 1 + 2 + 3 + ... + n - 1 = O(n^2)$.

Better solution:

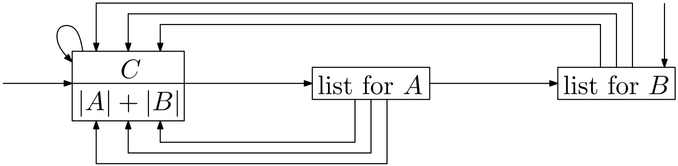
for each list, the head stores

- name of the set
- size of the set

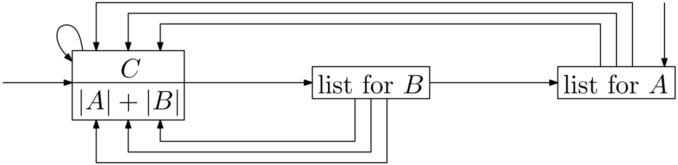
Find(x) takes $O(1)$ time, as before.

Union(A, B, C):

If $|A| \geq |B|$:



If $|A| < |B|$:



Time $= O(\min\{|A|, |B|\}) = O(\text{number of back-pointers that are changed})$

What is the total time for a sequence of $n-1$ **Union** operations:

$$\begin{aligned}\text{Total time} &= \text{total number of back-pointer changes} \\ &= \sum_{x=1}^{n} \text{total number of times that back}(x) \text{ is changed}\end{aligned}$$

What is the total time for a sequence of $n-1$ **Union** operations:

$$\begin{aligned}\text{Total time} &= \text{total number of back-pointer changes}\\ &= \sum_{x=1}^{n} \text{total number of times that back}(x) \text{ is changed}\end{aligned}$$

Consider an element x. How many times do we change back(x)?

What is the total time for a sequence of $n-1$ **Union** operations:

$$\begin{aligned}\text{Total time} &= \text{total number of back-pointer changes}\\ &= \sum_{x=1}^{n} \text{total number of times that back}(x) \text{ is changed}\end{aligned}$$

Consider an element x. How many times do we change back(x)?

Start: x is in a set of size 1.

What is the total time for a sequence of $n-1$ **Union** operations:

$$\begin{aligned}\text{Total time} &= \text{total number of back-pointer changes} \\ &= \sum_{x=1}^{n} \text{total number of times that back}(x) \text{ is changed}\end{aligned}$$

Consider an element x. How many times do we change back(x)?

Start: x is in a set of size 1.

First time that back(x) is changed:

the set containing x is merged with a set of size ≥ 1.

Hence, the new set containing x has size ≥ 2.

Second time that back(x) is changed:

the set containing x is merged with a set of size ≥ 2.

Hence, the new set continaing x has size ≥ 4.

Second time that back(x) is changed:

the set containing x is merged with a set of size ≥ 2.

Hence, the new set continaing x has size ≥ 4.

Third time that back(x) is changed:

the set containing x is merged with a set of size ≥ 4.

Hence, the new set continaing x has size ≥ 8.

Second time that back(x) is changed:

> the set containing x is merged with a set of size ≥ 2.
>
> Hence, the new set continaing x has size ≥ 4.

Third time that back(x) is changed:

> the set containing x is merged with a set of size ≥ 4.
>
> Hence, the new set continaing x has size ≥ 8.

etc.

Second time that back(x) is changed:

the set containing x is merged with a set of size ≥ 2.

Hence, the new set contining x has size ≥ 4.

Third time that back(x) is changed:

the set containing x is merged with a set of size ≥ 4.

Hence, the new set contining x has size ≥ 8.

etc.

Since there are n elements, back(x) is changed $\leq \log_2(n)$ times.

Second time that back(x) is changed:

the set containing x is merged with a set of size ≥ 2.

Hence, the new set continaing x has size ≥ 4.

Third time that back(x) is changed:

the set containing x is merged with a set of size ≥ 4.

Hence, the new set continaing x has size ≥ 8.

etc.

Since there are n elements, back(x) is changed $\leq \log_2(n)$ times.

Therefore, the total time for $n-1$ **Union** operations $= O(n \log(n))$.

Second time that back(x) is changed:

the set containing x is merged with a set of size ≥ 2.

Hence, the new set continaing x has size ≥ 4.

Third time that back(x) is changed:

the set containing x is merged with a set of size ≥ 4.

Hence, the new set continaing x has size ≥ 8.

etc.

Since there are n elements, back(x) is changed $\leq \log_2(n)$ times.

Therefore, the total time for $n-1$ **Union** operations $= O(n \log(n))$.

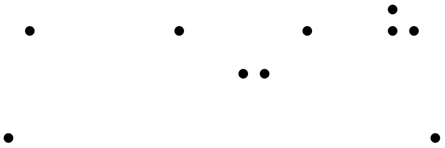
Conclusion: Any sequence of $n-1$ **Union** and m **Find** operations takes $O(m + n \log(n))$ time.

)

Kruskal Algorithm (1956)

Approach : Maintain a forest. In each step, add an edge of minimum weight that does not create a cycle.

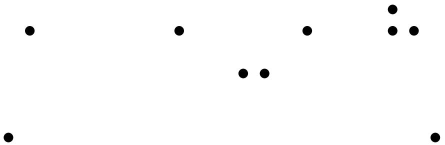
Start : At the beginning, each vertex is a (trivial) tree.



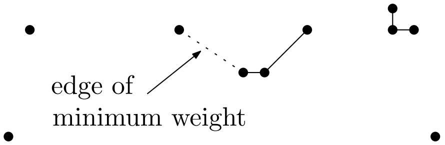
Kruskal Algorithm (1956)

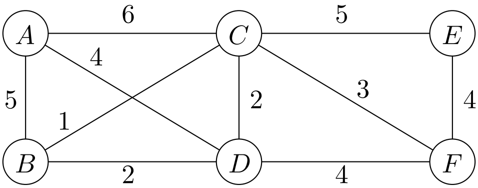
Approach : Maintain a forest. In each step, add an edge of minimum weight that does not create a cycle.

Start : At the beginning, each vertex is a (trivial) tree.



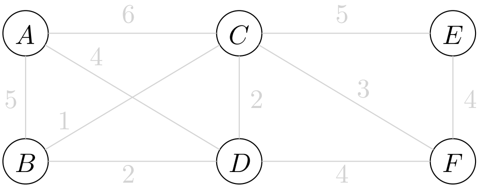
One Iteration : Combine two trees using an edge of minimum weight.

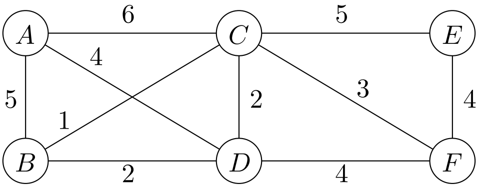




Sor the edges by weight:

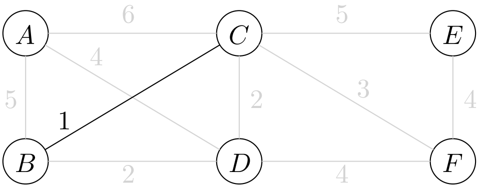
$$BC, BD, CD, CF, AD, DF, EF, AB, CE, AC$$

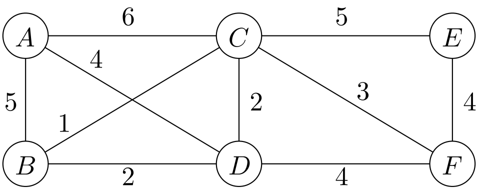




Sor the edges by weight:

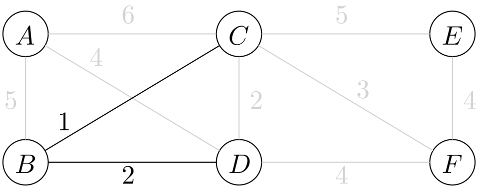
$$BC, BD, CD, CF, AD, DF, EF, AB, CE, AC$$

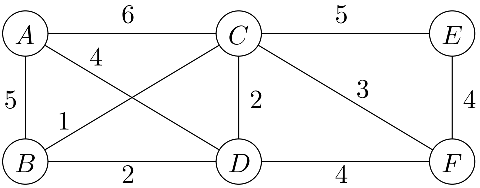




Sor the edges by weight:

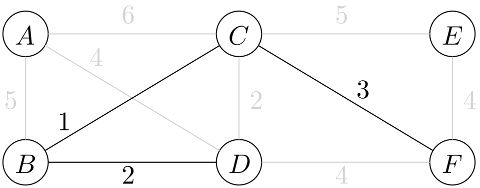
$$BC, BD, CD, CF, AD, DF, EF, AB, CE, AC$$

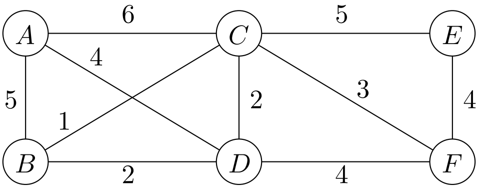




Sor the edges by weight:

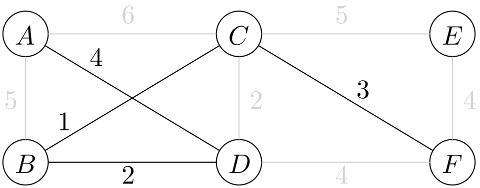
$$BC, BD, CD, CF, AD, DF, EF, AB, CE, AC$$

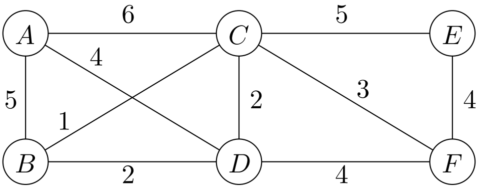




Sor the edges by weight:

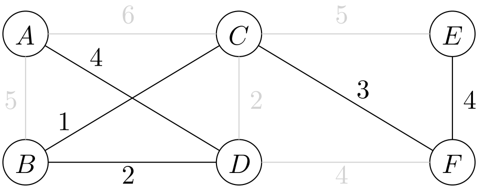
$$BC, BD, CD, CF, AD, DF, EF, AB, CE, AC$$





Sor the edges by weight:

$$BC, BD, CD, CF, AD, DF, EF, AB, CE, AC$$



Total weight: 14

Algorithm *Kruskal*(G)

Input: $G = (V, E)$, where $V = \{x_1, x_2, \ldots, x_n\}$ and $m = |E|$.
Output: A minimum spanning tree of G.

Sort the edges of E by weight using Merge Sort: $e_1, e_2, \ldots, e_m$
for $i = 1$ to n **do**
    $V_i = \{x_i\}$
end for
$X = \{\ \}$
for $k = 1$ to m **do**
    let u_k and v_k be the vertices of e_k.
    let i be the index such that $u_k \in V_i$
    let j be the index such that $v_k \in V_j$
    if $i \neq j$ **then**
        $V_i = V_i \cup V_j$
        $X = X \cup \{\{u_k, v_k\}\}$
    end if
end for
return X

Running time:

- Sorting: $O(m \log(m)) = O(m \log(n))$ time (do you see why?)

Running time:

- Sorting: $O(m\log(m)) = O(m\log(n))$ time (do you see why?)
- First For-loop: $O(n)$ time

Running time:

- Sorting: $O(m\log(m)) = O(m\log(n))$ time (do you see why?)
- First For-loop: $O(n)$ time
- Second For-loop:
  - Store X in a linked list. Total time to maintain this list: $O(n)$ time
  - Store the sets V_i using the Union-Find data structure.

  In this second For-Loop, we do
  - $2m$ **Find** operations
  - $n-1$ **Union** operations

  So in total for the second For-Loop:

$$O(n) + O(m + n\log(n)) \text{ time}$$

Running time:

- Sorting: $O(m\log(m)) = O(m\log(n))$ time (do you see why?)
- First For-loop: $O(n)$ time
- Second For-loop:
  - Store X in a linked list. Total time to maintain this list: $O(n)$ time
  - Store the sets V_i using the Union-Find data structure.

  In this second For-Loop, we do
  - $2m$ **Find** operations
  - $n-1$ **Union** operations

  So in total for the second For-Loop:

$$O(n) + O(m + n\log(n)) \text{ time}$$

So the total time is

$$O(m\log(n)) + O(n) + O(m + n\log(n)) = O(m\log(n))$$

Do you see why?

Conclusion: Kruskal computes an MST in $O(m \log(n))$ time.