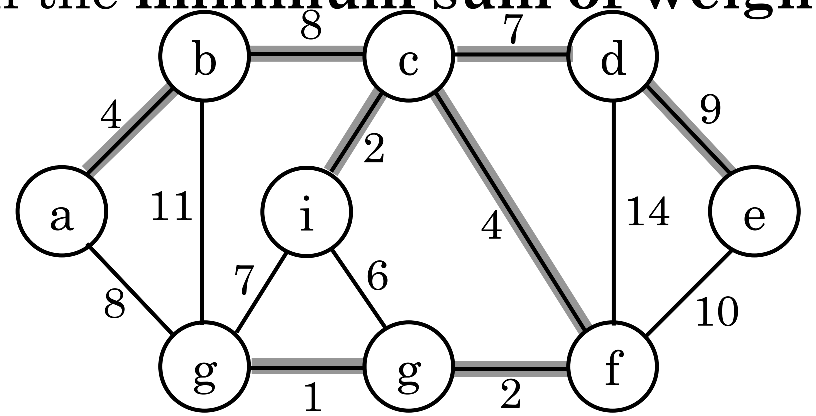
Graph Theory
Minimum Spanning Tree

Design and Analysis of Algorithms

Fall 2022

Minimum Spanning Trees

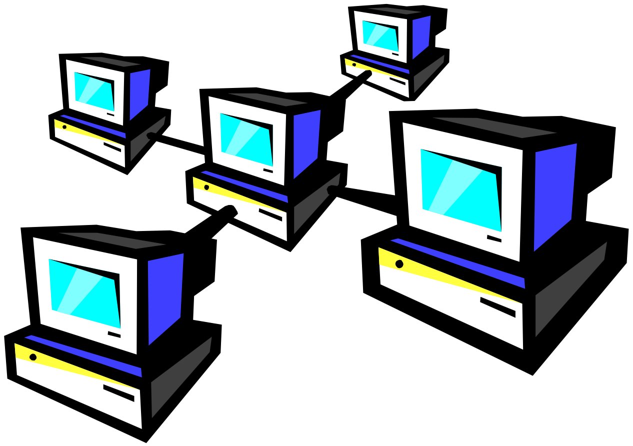
- Spanning Tree
  - A tree (i.e., connected, acyclic graph) which contains all the vertices of the graph
- Minimum Spanning Tree
  - Spanning tree with the **minimum sum of weights**

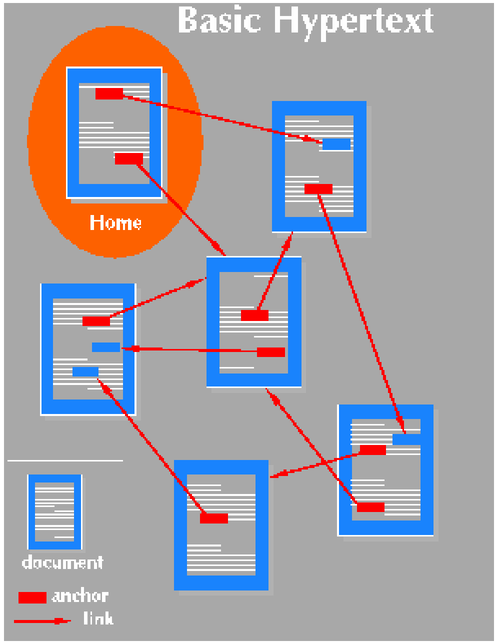


- Spanning forest
  - If a graph is not connected, then there is a spanning tree for each connected component of the graph

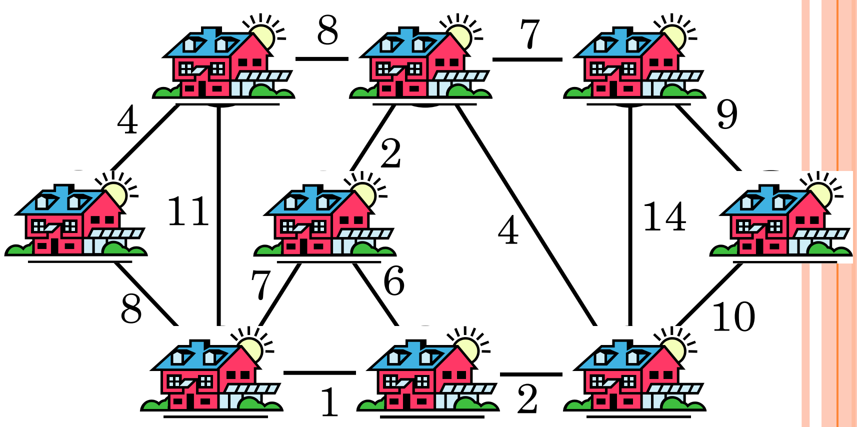
Applications of MST

- Find the least expensive way to connect a set of cities, terminals, computers, etc.





Example



Problem

- A town has a set of houses and a set of roads
- A road connects 2 and only 2 houses
- A road connecting houses u and v has a repair cost w(u, v)

Goal: Repair enough (and no more) roads such that:

1. Everyone stays connected

   i.e., can reach every house from all other houses

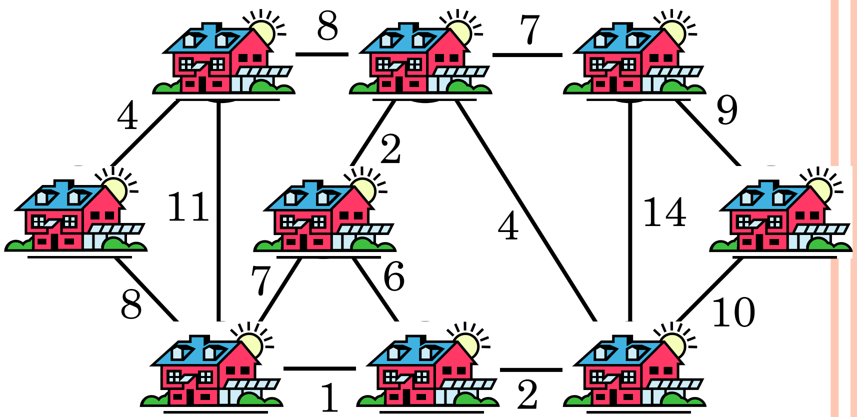
2. Total repair cost is minimum

MINIMUM SPANNING TREES

- A connected, undirected graph:
  - Vertices = houses, Edges = roads
- A **weight** $w(u, v)$ on each edge $(u, v) \in E$

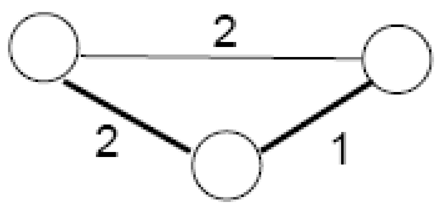
Find $T \subseteq E$ such that:

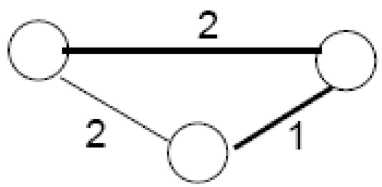
1. T connects all vertices
2. $w(T) = \Sigma_{(u,v)\in T}\, w(u, v)$ is minimized



Properties of Minimum Spanning Trees

- Minimum spanning tree is **not** unique





- MST has no cycles – see why:
  - We can take out an edge of a cycle, and still have the vertices connected while reducing the cost
- # of edges in a MST:
  - |V| - 1

Minimum spanning trees

- A ***spanning tree*** of an undirected connected graph is its connected acyclic subgraph (i.e., a tree) that contains all the vertices of the graph.
- If such a graph has weights assigned to its edges, a ***minimum spanning tree*** is its spanning tree of the smallest weight, where the ***weight*** of a tree is defined as the sum of the
- weights on all its edges.
- The ***minimum spanning tree problem*** is the problem of finding a minimum spanning tree for a given weighted connected graph.

MINIMUM SPANNING TREES

Input: A connected, undirected graph $G = (V, E)$ with weight function $w : E \to \mathbb{R}$.

- For simplicity, assume that all edge weights are distinct. (CLRS covers the general case.)

MINIMUM SPANNING TREES

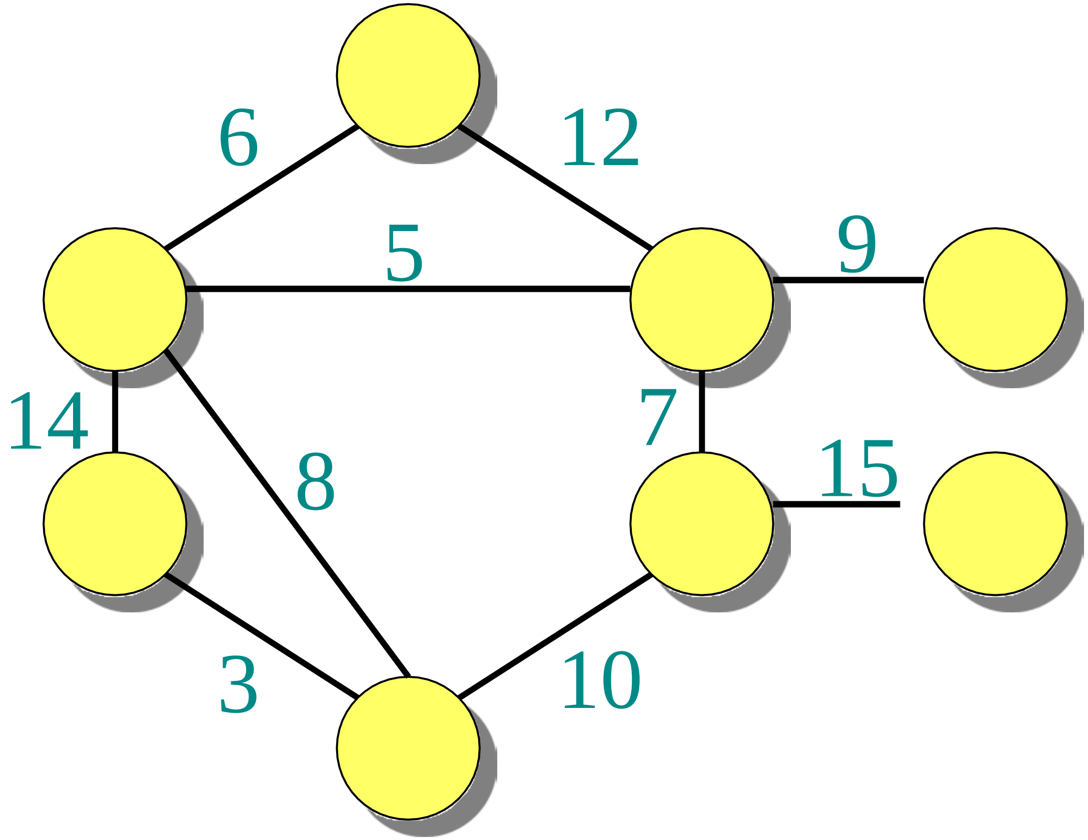
Input: A connected, undirected graph $G = (V, E)$ with weight function $w : E \to \mathbb{R}$.

- For simplicity, assume that all edge weights are distinct. (CLRS covers the general case.)

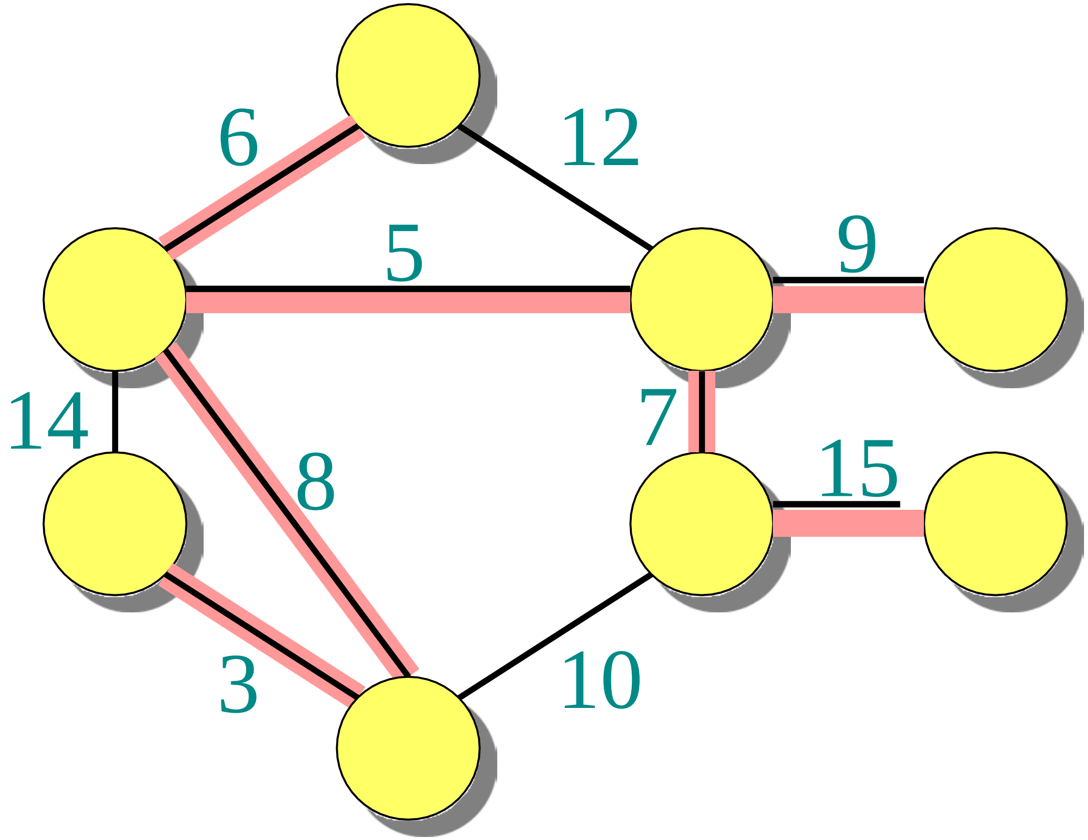
Output: A ***spanning tree*** T — a tree that connects all vertices — of minimum weight:

$$w(T) = \sum_{(u,v) \in T} w(u,v).$$

Example of MST



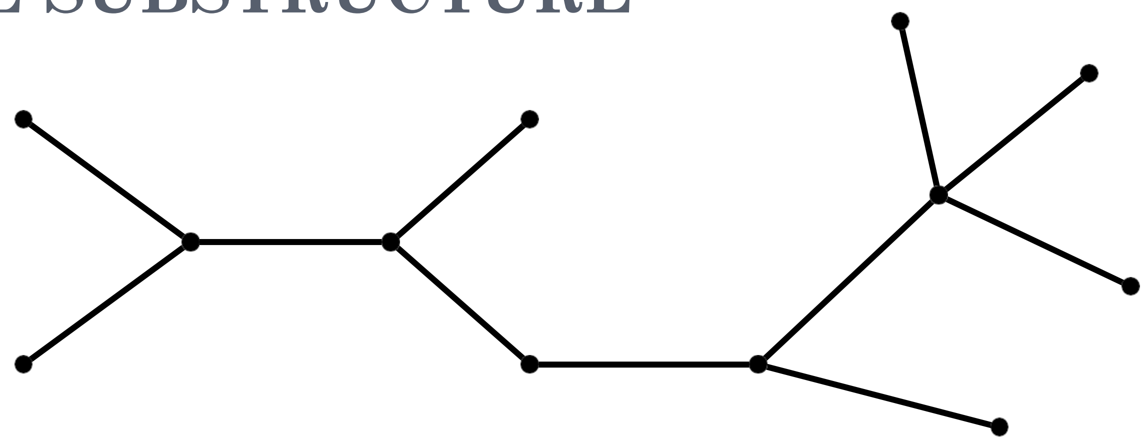
EXAMPLE OF MST



Optimal substructure

MST *T*:

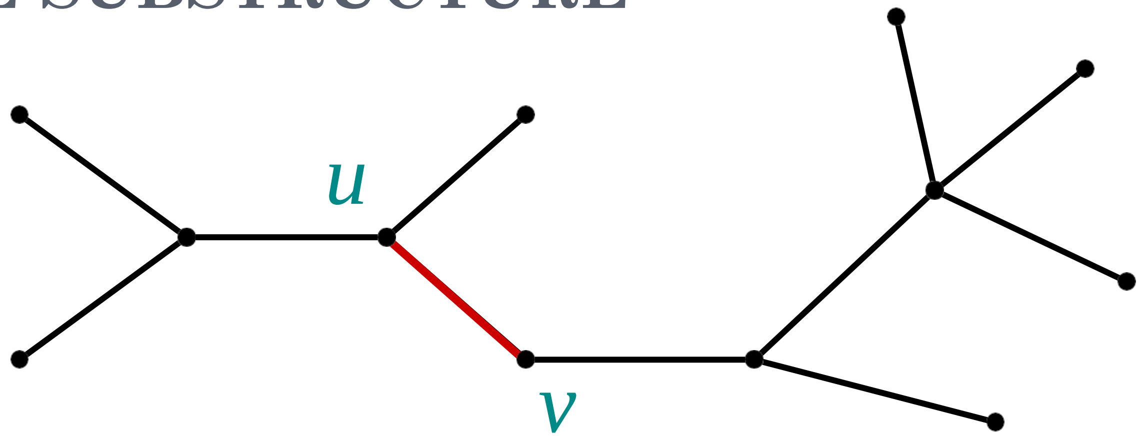
(Other edges of *G* are not shown.)



OPTIMAL SUBSTRUCTURE

MST T:

(Other edges of G are not shown.)

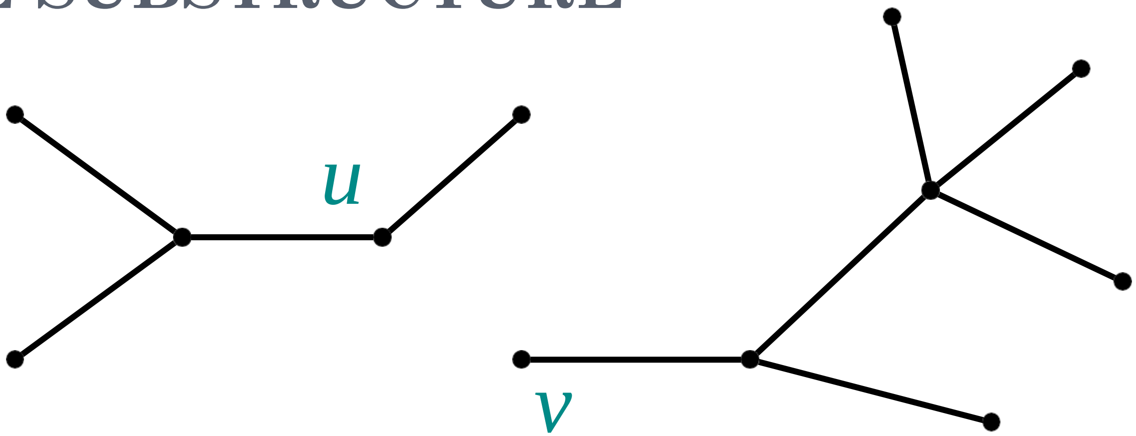


Remove any edge $(u, v) \in T$.

Optimal substructure

MST T:

(Other edges of G are not shown.)

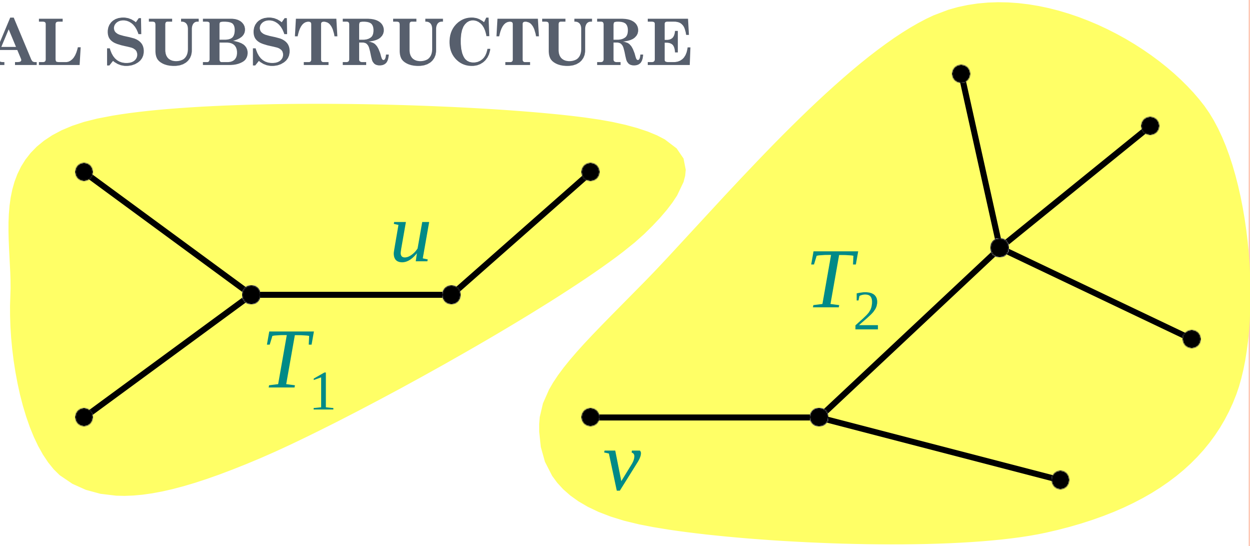


Remove any edge $(u, v) \in T$.

OPTIMAL SUBSTRUCTURE

MST T:

(Other edges of G are not shown.)



Remove any edge $(u, v) \in T$. Then, T is partitioned into two subtrees T_1 and T_2.

Hallmark for "greedy" algorithms

Greedy-choice property

A locally optimal choice is globally optimal.

```
GENERIC-MST(G, w)
A = ∅
while A does not form a spanning tree
    find an edge (u, v) that is safe for A
    A = A ∪ {(u, v)}
return A
```

- The generic method manages a set of edges A, maintaining the following loop invariant:
- **Prior to each iteration, A is a subset of some minimum spanning tree.**
- At each step, we determine an edge {u, v} that we can add to A without violating this invariant, in the sense that A U {u, v] is also a subset of a minimum spanning tree.
- We call such an edge a ***safe edge*** for A, since we can add it safely to A while maintaining the invariant

Hallmark for "greedy" algorithms

Greedy-choice property

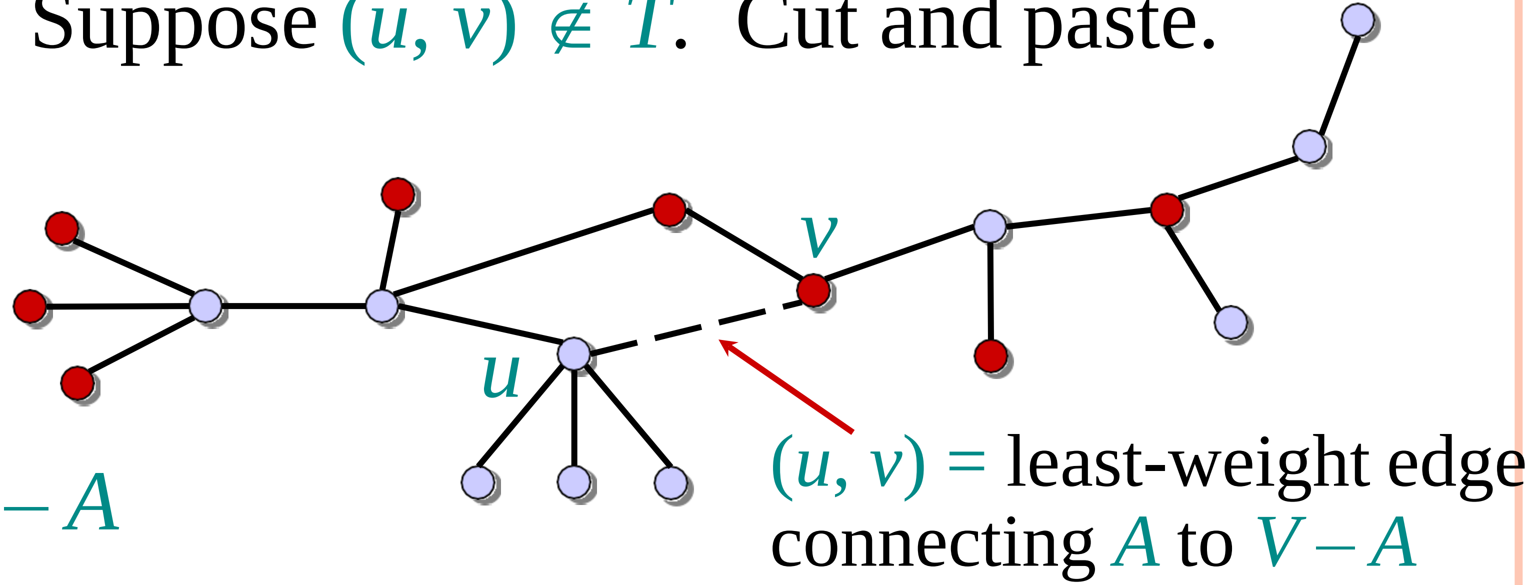
A locally optimal choice is globally optimal.

Theorem. Let T be the MST of $G = (V, E)$, and let $A \subseteq V$. Suppose that $(u, v) \in E$ is the least-weight edge connecting A to $V - A$. Then, $(u, v) \in T$.

Proof of theorem

Proof. Suppose $(u, v) \notin T$. Cut and paste.

T:

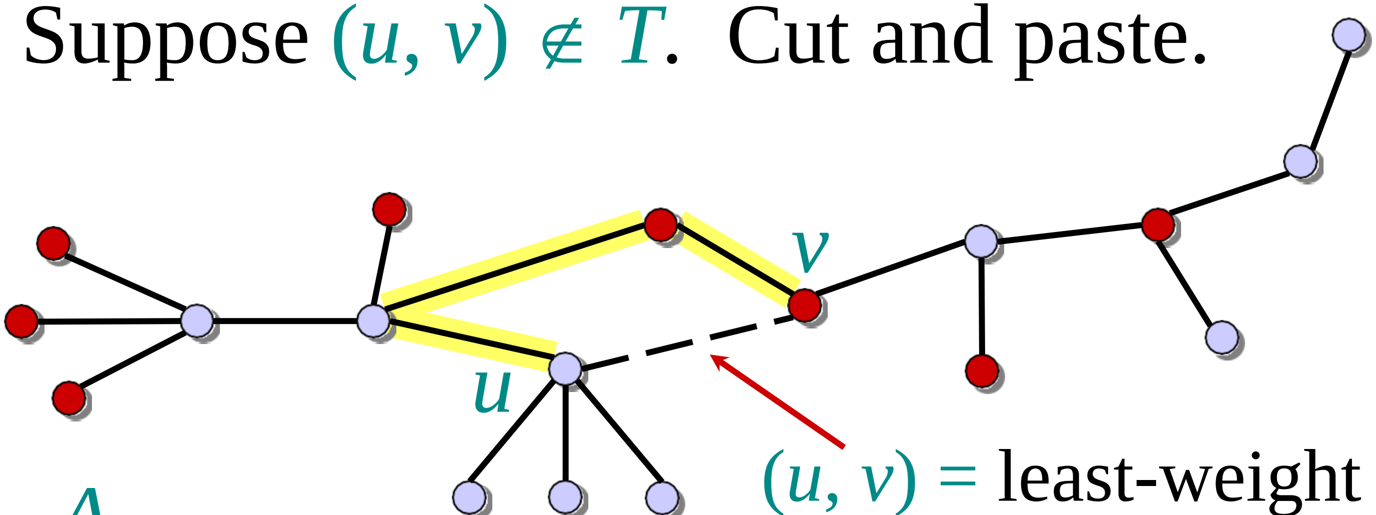


● $\in A$
● $\in V - A$

Proof of theorem

Proof. Suppose $(u, v) \notin T$. Cut and paste.

T:



- $\in A$
- $\in V - A$

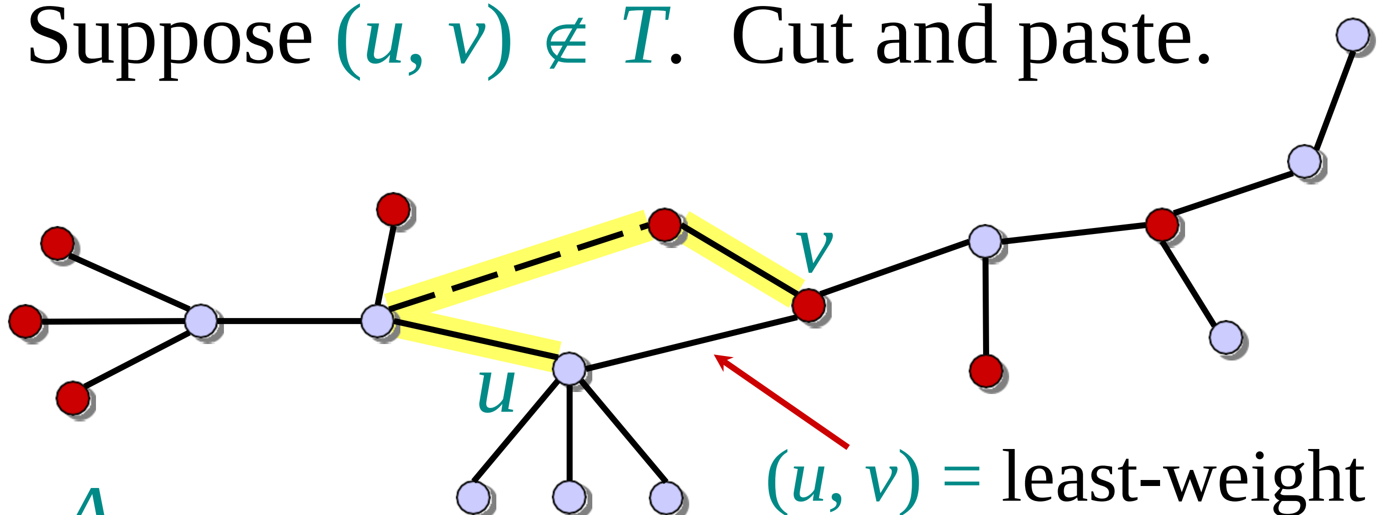
(u, v) = least-weight edge connecting A to $V - A$

Consider the unique simple path from u to v in T.

Proof of theorem

Proof. Suppose $(u, v) \notin T$. Cut and paste.

T:



$\in A$

$\in V - A$

(u, v) = least-weight edge connecting A to $V - A$

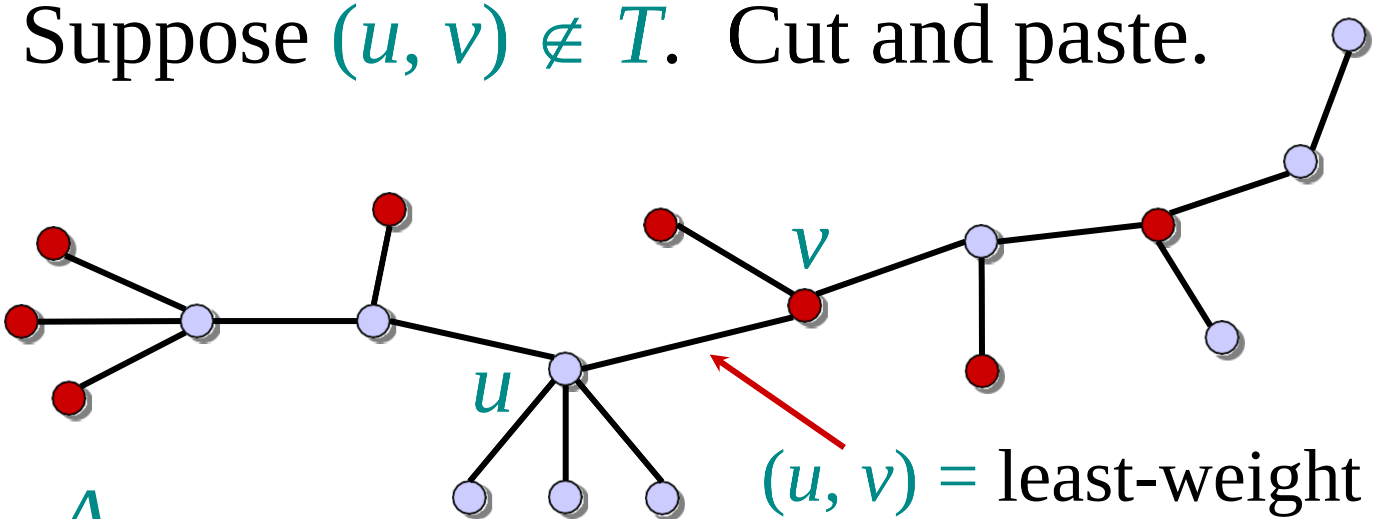
Consider the unique simple path from u to v in T.

Swap (u, v) with the first edge on this path that connects a vertex in A to a vertex in $V - A$.

Proof of theorem

Proof. Suppose $(u, v) \notin T$. Cut and paste.

T:



● $\in A$

● $\in V - A$

Consider the unique simple path from u to v in T.

Swap (u, v) with the first edge on this path that connects a vertex in A to a vertex in $V - A$.

A lighter-weight spanning tree than T results.

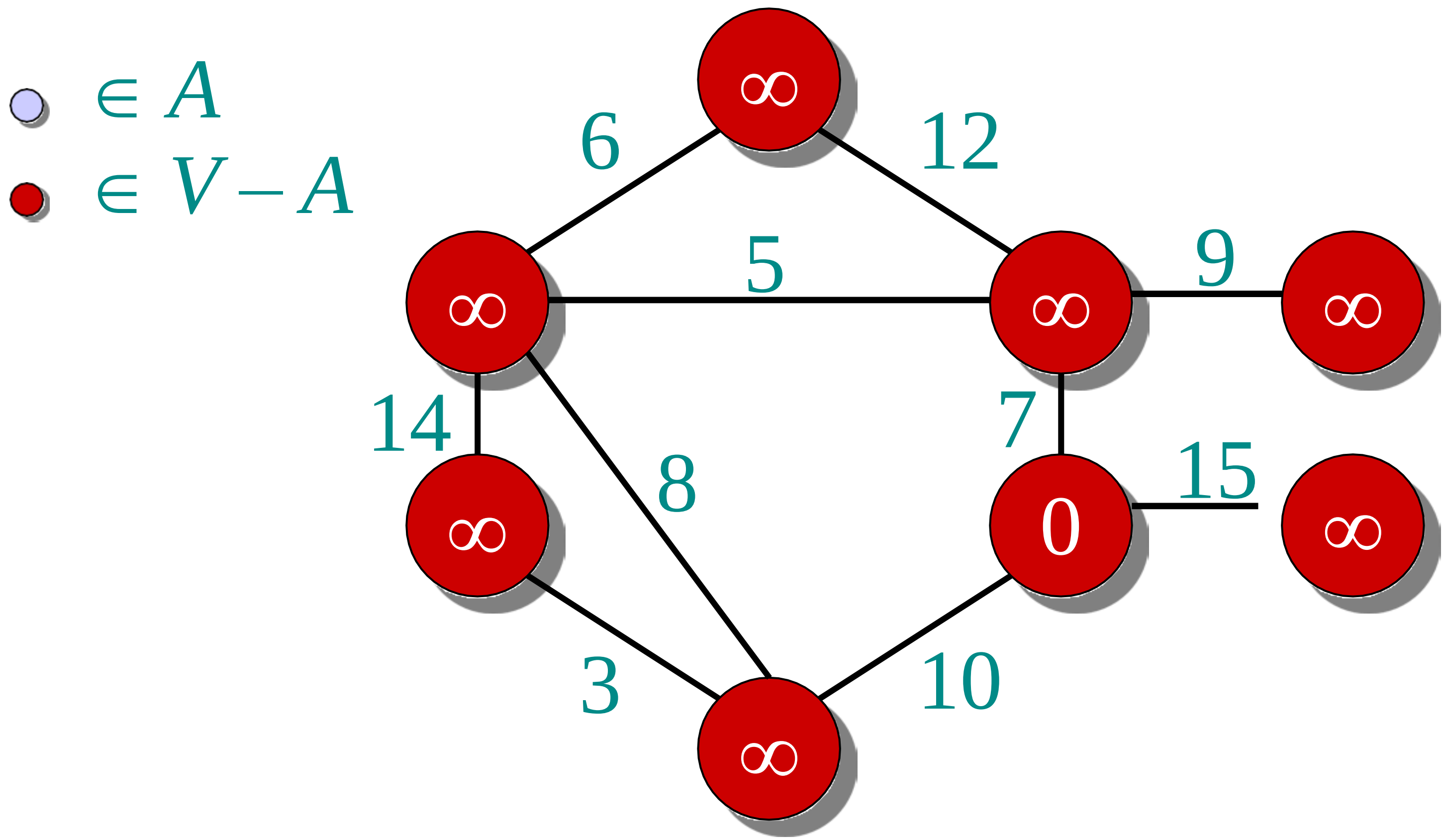
PRIM'S ALGORITHM

IDEA: Maintain $V - A$ as a priority queue Q. Key each vertex in Q with the weight of the least-weight edge connecting it to a vertex in A.

$Q \leftarrow V$
$key[v] \leftarrow \infty$ for all $v \in V$
$key[s] \leftarrow 0$ for some arbitrary $s \in V$
while $Q \neq \emptyset$
    do $u \leftarrow$ EXTRACT-MIN(Q)
        for each $v \in Adj[u]$
            do if $v \in Q$ and $w(u, v) < key[v]$
                then $key[v] \leftarrow w(u, v)$ ⊳ DECREASE-KEY
                    $\pi[v] \leftarrow u$

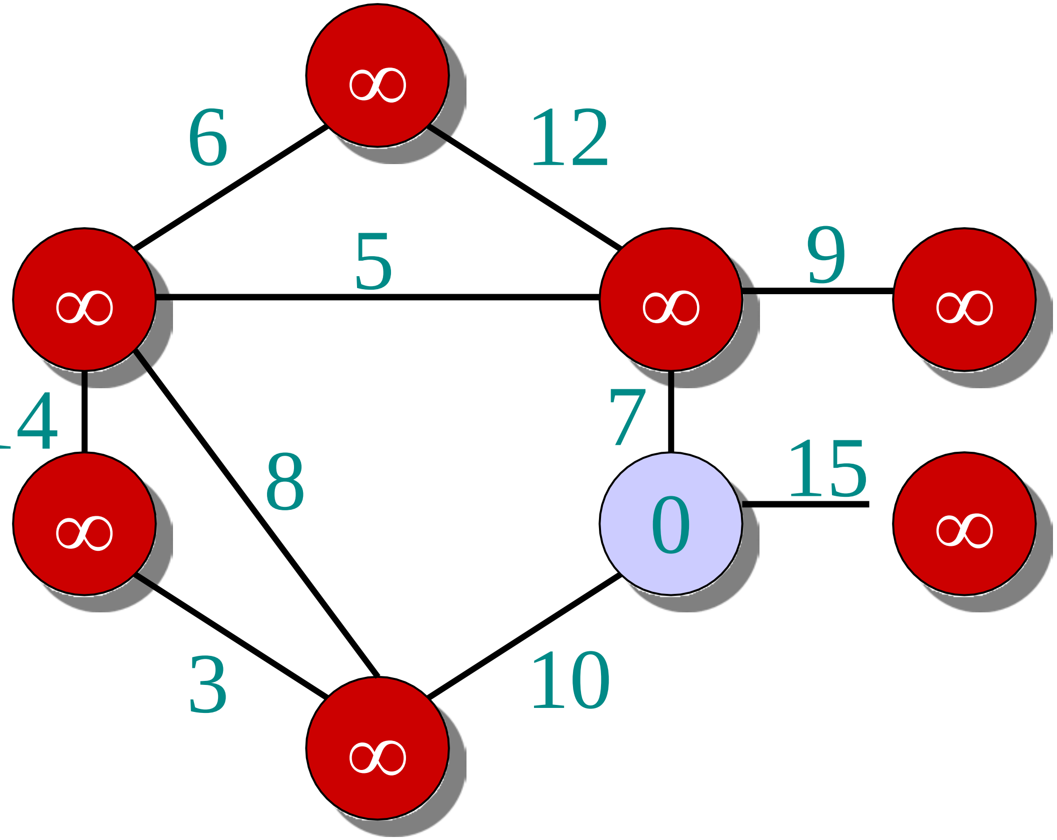
At the end, $\{(v, \pi[v])\}$ forms the MST.

Example of Prim's algorithm

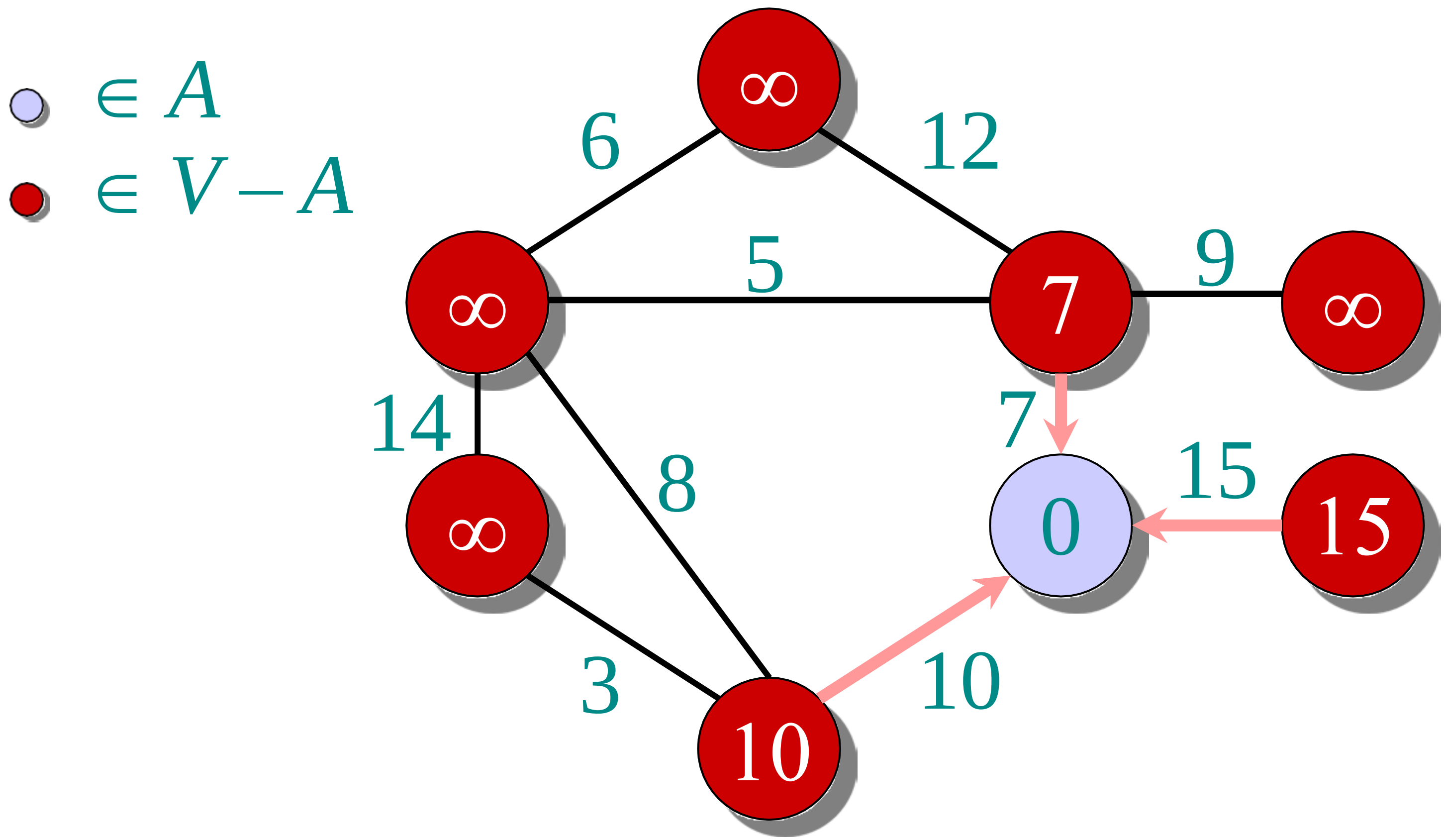


Example of Prim's algorithm

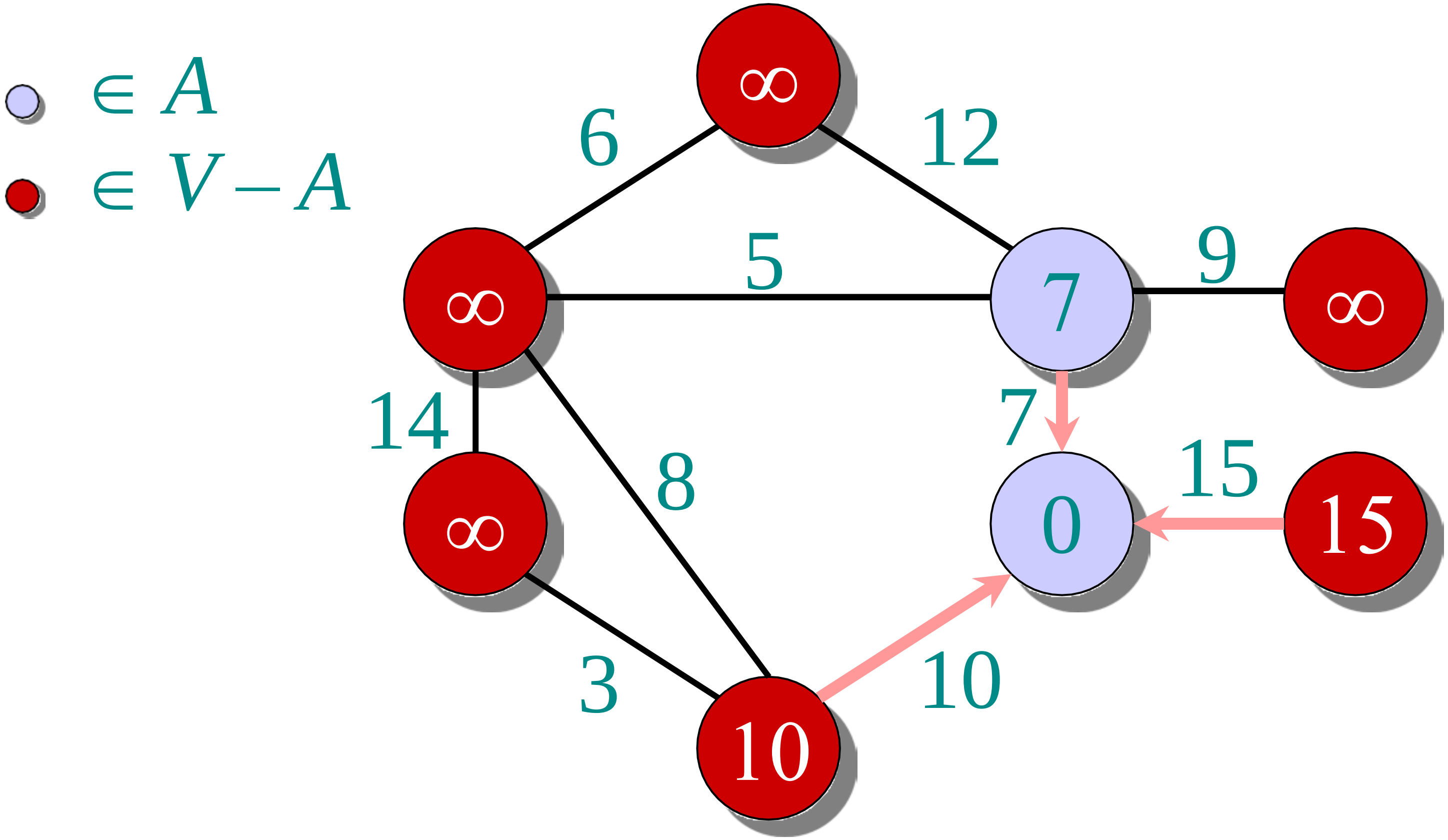
- $\in A$
- $\in V - A$



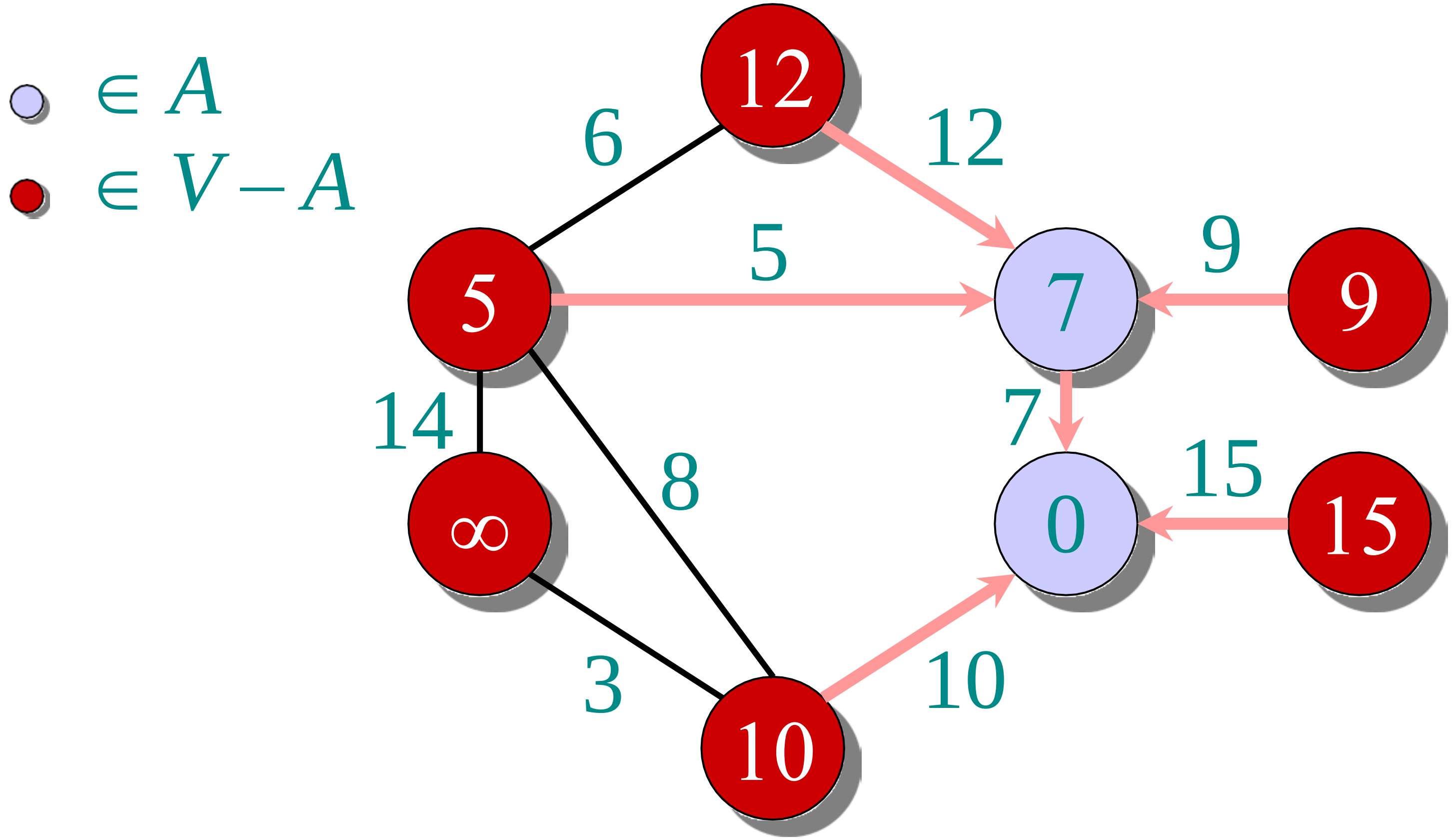
EXAMPLE OF PRIM'S ALGORITHM



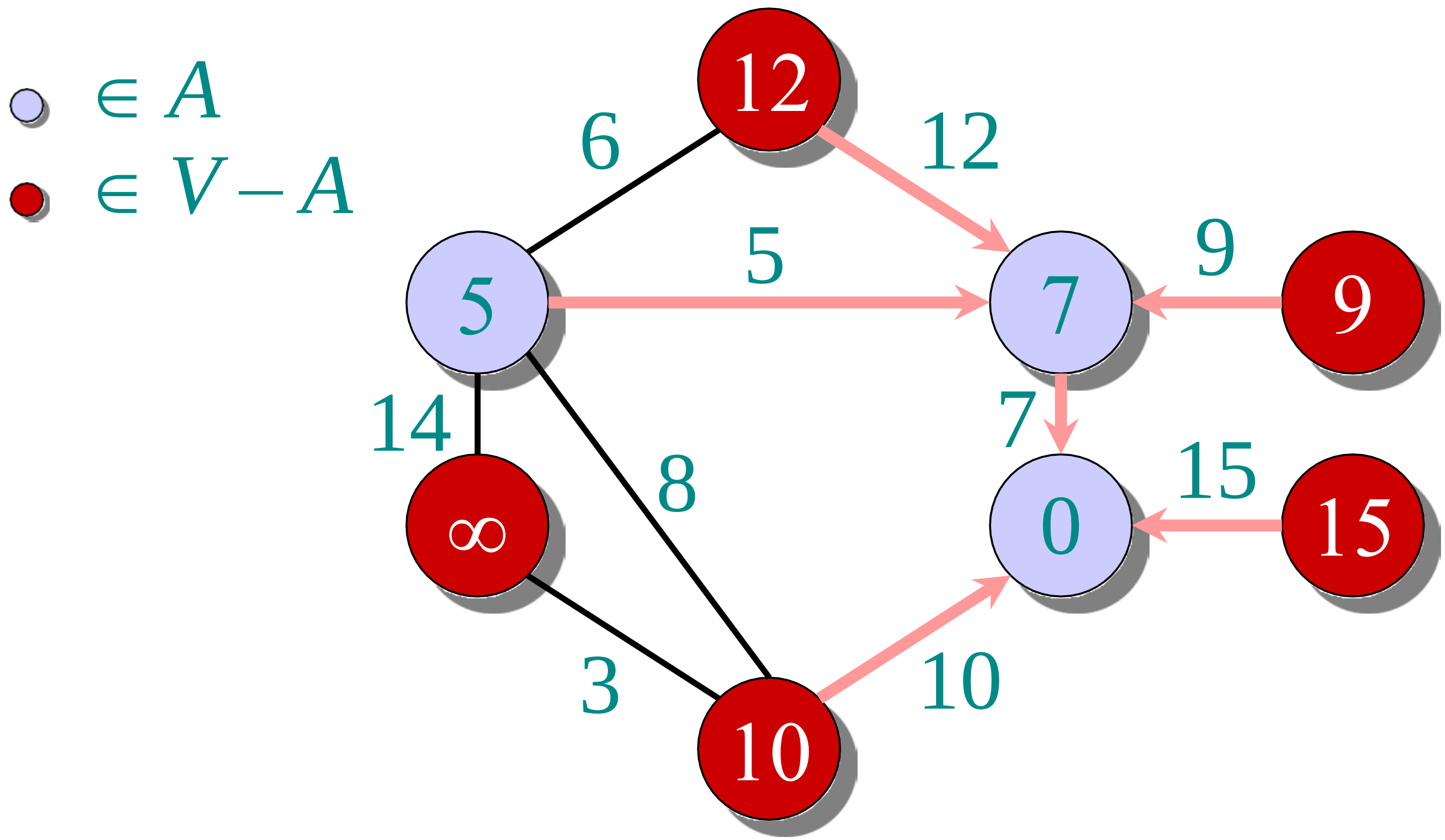
EXAMPLE OF PRIM'S ALGORITHM



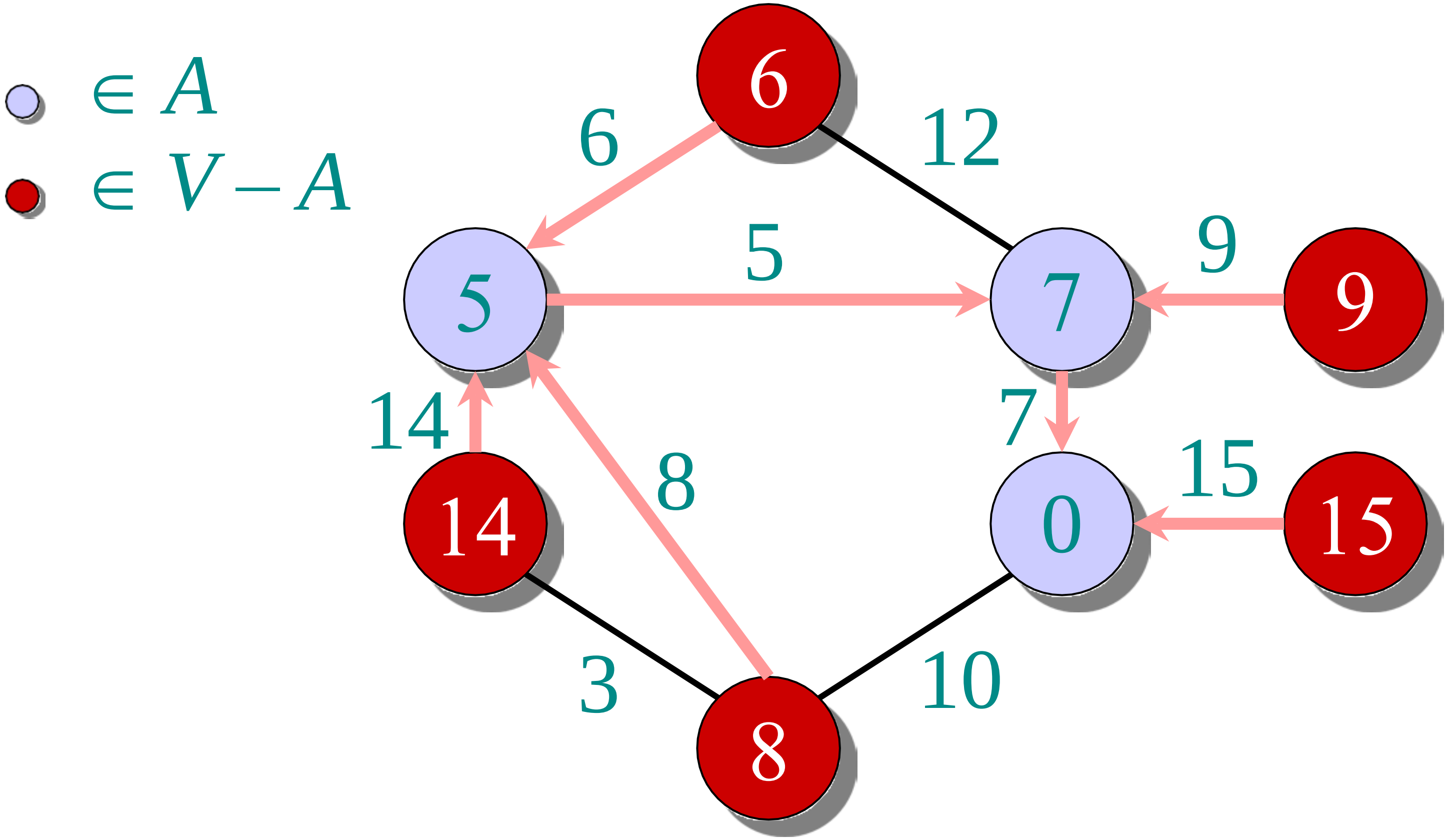
Example of Prim's algorithm



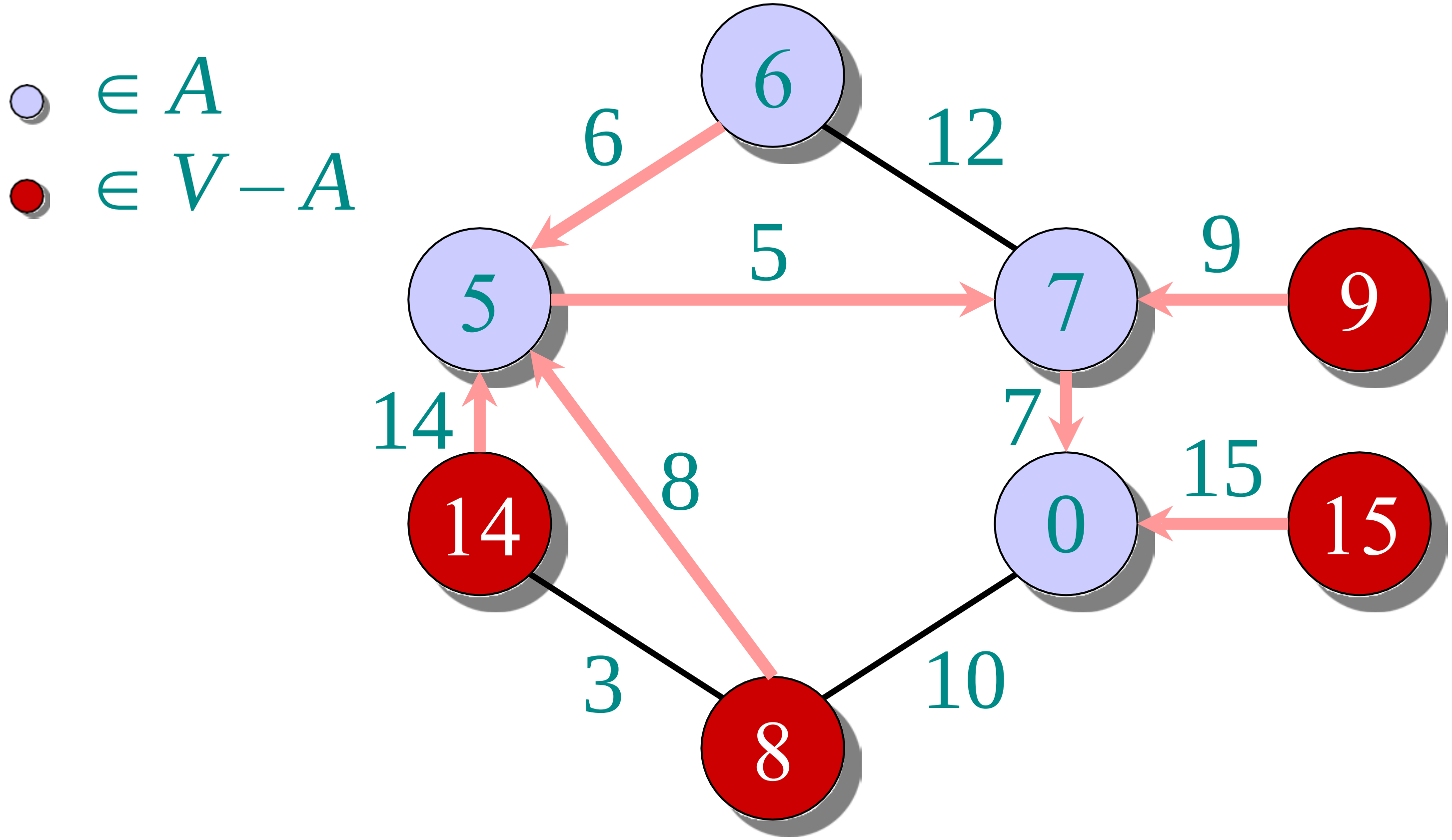
Example of Prim's algorithm



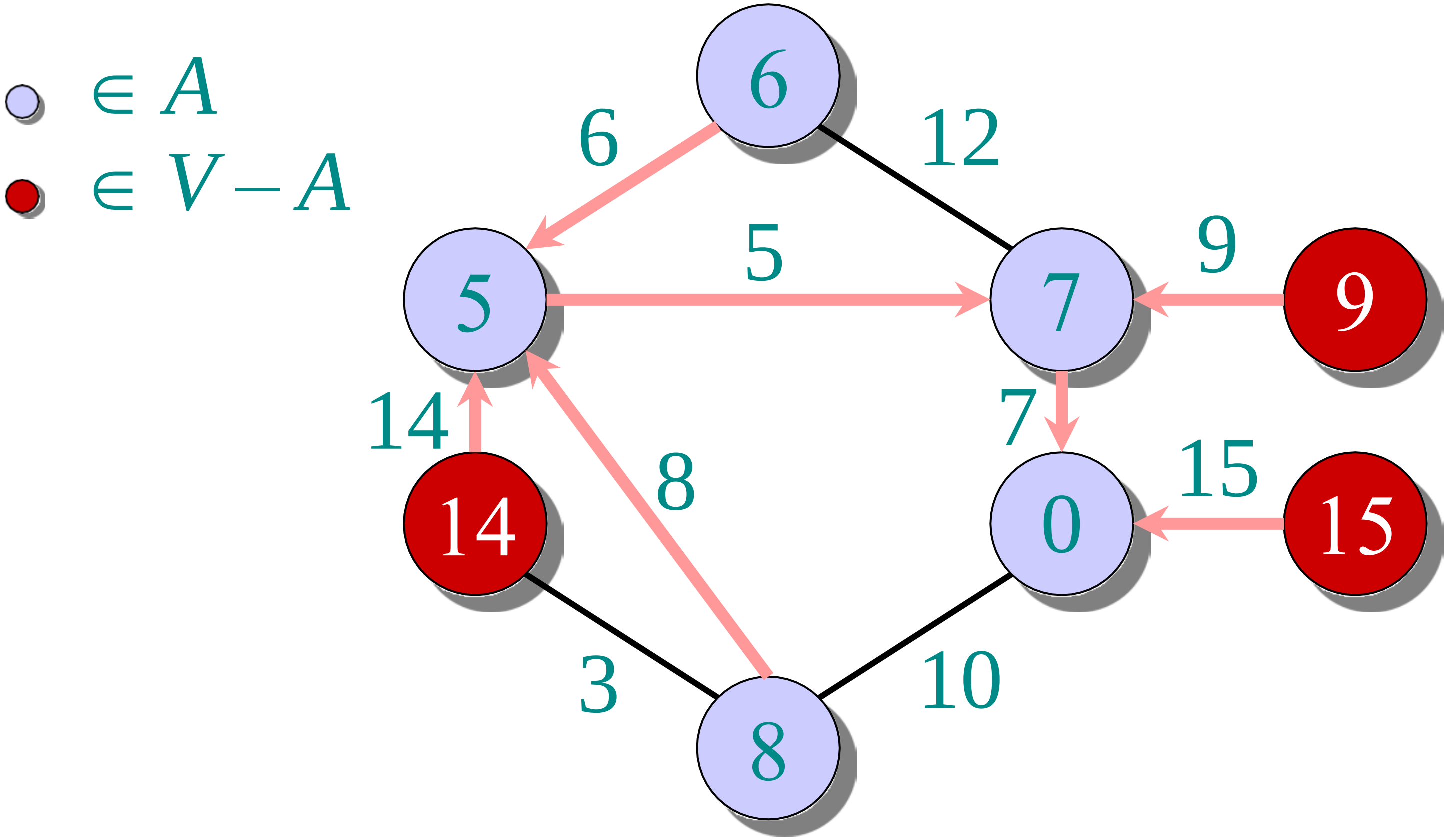
EXAMPLE OF PRIM'S ALGORITHM



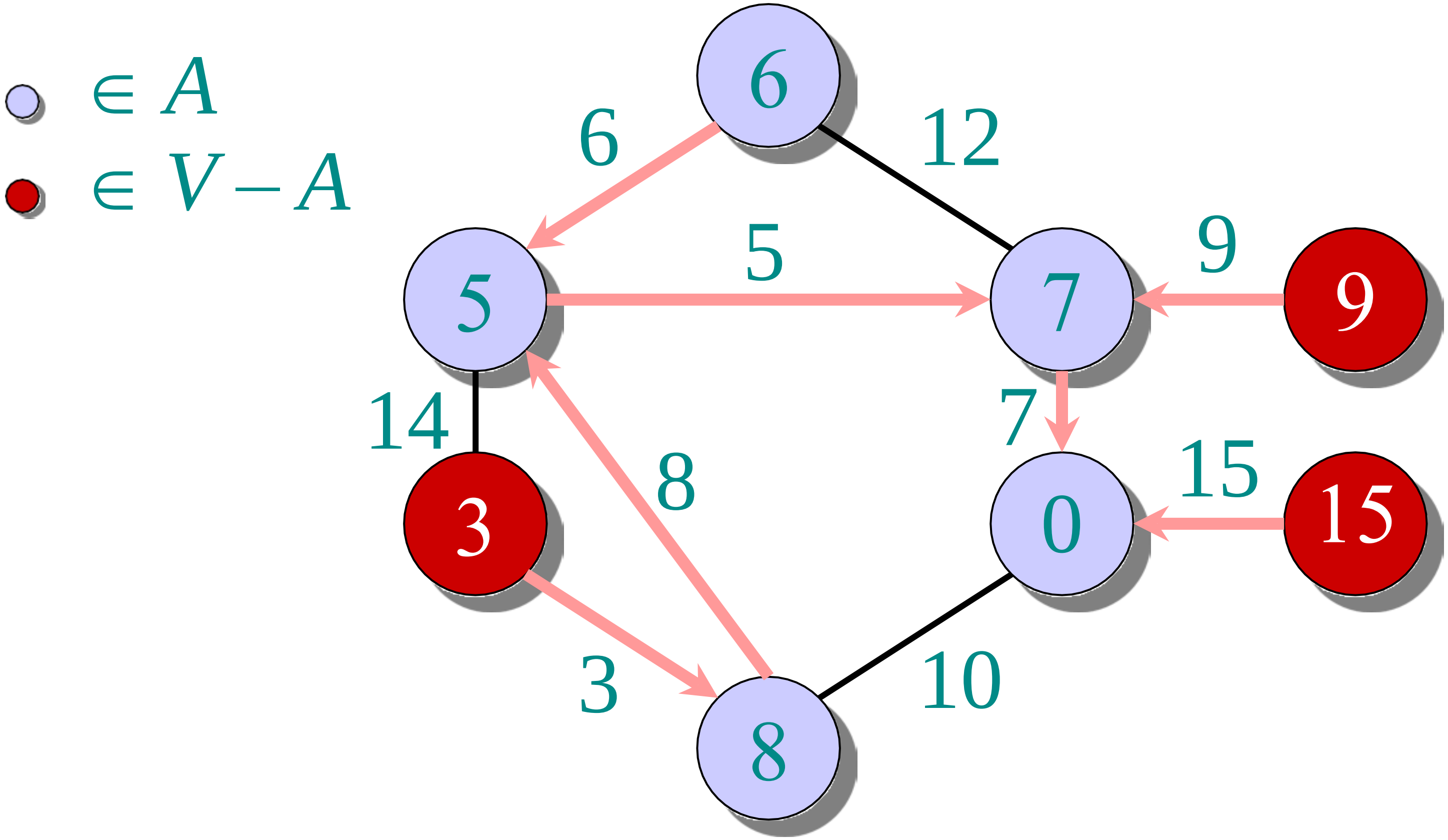
Example of Prim's algorithm



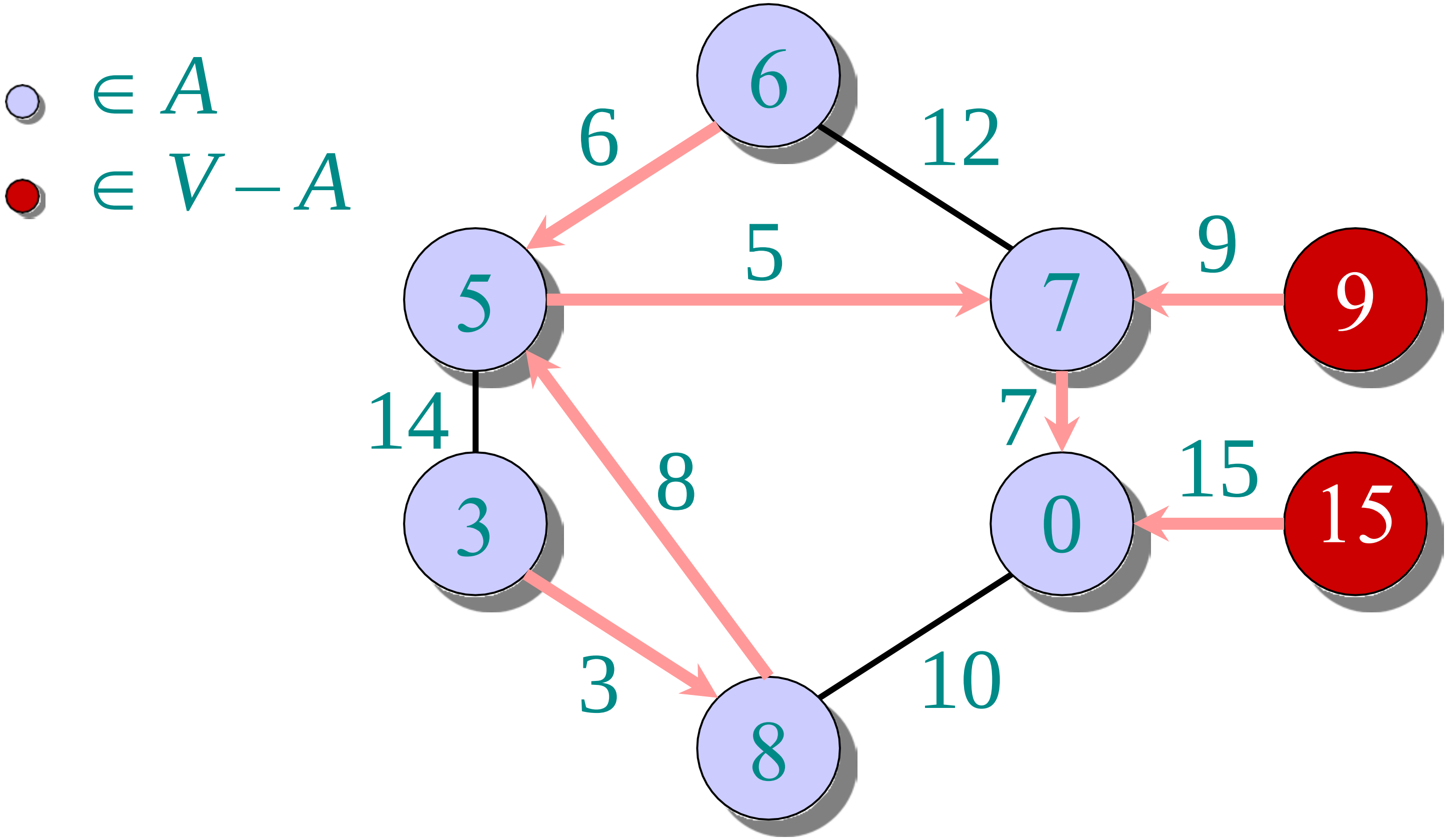
Example of Prim's algorithm



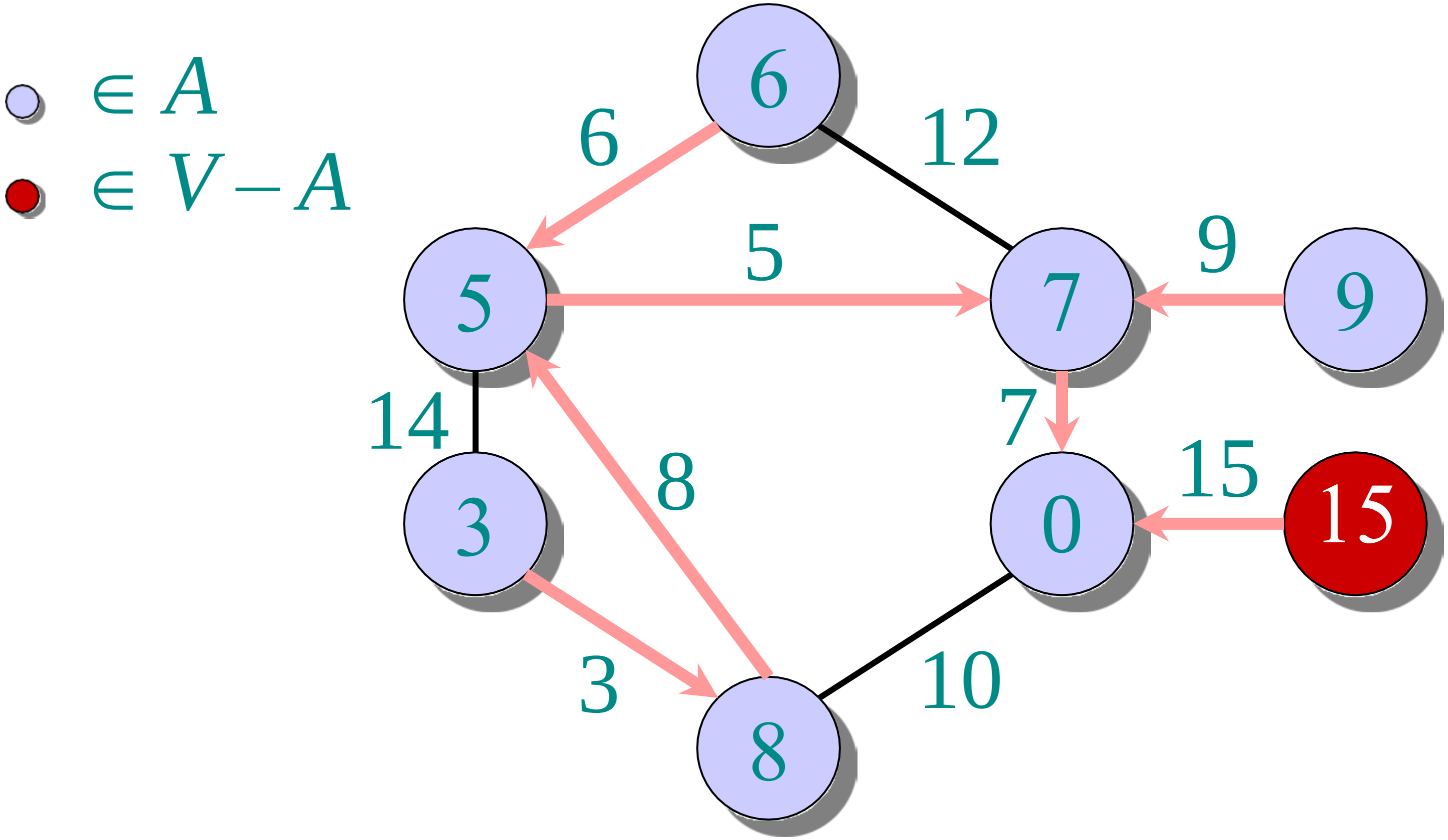
Example of Prim's algorithm



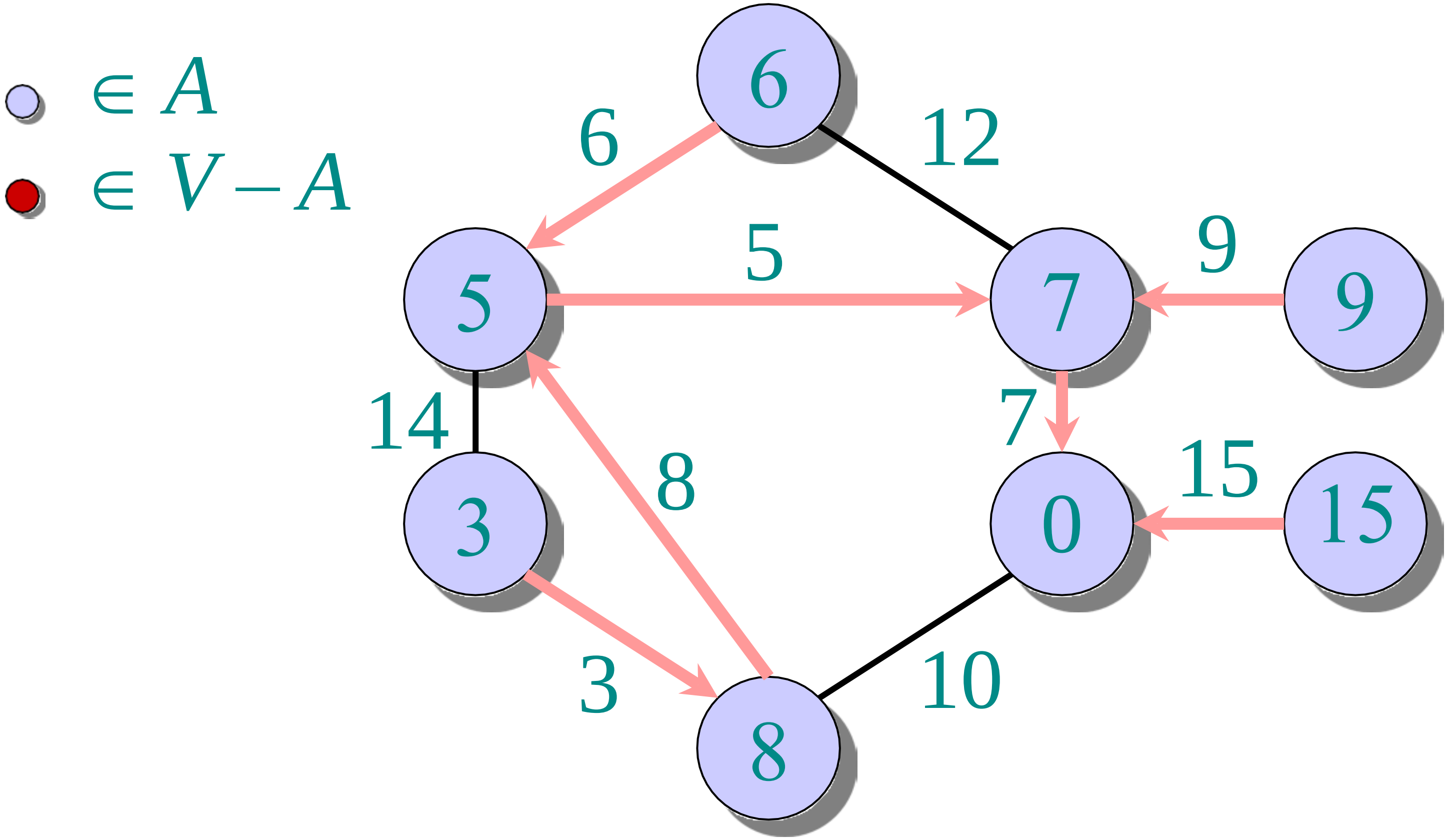
EXAMPLE OF PRIM'S ALGORITHM



EXAMPLE OF PRIM'S ALGORITHM



EXAMPLE OF PRIM'S ALGORITHM



Analysis of Prim

$Q \leftarrow V$
$key[v] \leftarrow \infty$ for all $v \in V$
$key[s] \leftarrow 0$ for some arbitrary $s \in V$
while $Q \neq \emptyset$
  do $u \leftarrow$ EXTRACT-MIN(Q)
    for each $v \in Adj[u]$
      do if $v \in Q$ and $w(u, v) < key[v]$
        then $key[v] \leftarrow w(u, v)$
          $\pi[v] \leftarrow u$

Analysis of Prim

$\Theta(V)$ total $\left\{\begin{array}{l} Q \leftarrow V \\ key[v] \leftarrow \infty \text{ for all } v \in V \\ key[s] \leftarrow 0 \text{ for some arbitrary } s \in V \end{array}\right.$

while $Q \neq \emptyset$
  do $u \leftarrow$ EXTRACT-MIN(Q)
    for each $v \in Adj[u]$
      do if $v \in Q$ and $w(u, v) < key[v]$
        then $key[v] \leftarrow w(u, v)$
          $\pi[v] \leftarrow u$

Analysis of Prim

$\Theta(V)$ total $\left\{ \begin{array}{l} Q \leftarrow V \\ key[v] \leftarrow \infty \text{ for all } v \in V \\ key[s] \leftarrow 0 \text{ for some arbitrary } s \in V \end{array} \right.$

$|V|$ times $\left\{ \begin{array}{l} \textbf{while } Q \neq \emptyset \\ \quad \textbf{do } u \leftarrow \text{EXTRACT-MIN}(Q) \\ \qquad \textbf{for} \text{ each } v \in Adj[u] \\ \qquad\quad \textbf{do if } v \in Q \text{ and } w(u, v) < key[v] \\ \qquad\qquad \textbf{then } key[v] \leftarrow w(u, v) \\ \qquad\qquad\qquad \pi[v] \leftarrow u \end{array} \right.$

ANALYSIS OF PRIM

$\Theta(V)$ total

$Q \leftarrow V$
$key[v] \leftarrow \infty$ for all $v \in V$
$key[s] \leftarrow 0$ for some arbitrary $s \in V$

$|V|$ times

while $Q \neq \emptyset$
  do $u \leftarrow$ EXTRACT-MIN(Q)

$degree(u)$ times

    for each $v \in Adj[u]$
      do if $v \in Q$ and $w(u, v) < key[v]$
        then $key[v] \leftarrow w(u, v)$
          $\pi[v] \leftarrow u$

ANALYSIS OF PRIM

$\Theta(V)$ total $\left\{ \begin{array}{l} Q \leftarrow V \\ key[v] \leftarrow \infty \text{ for all } v \in V \\ key[s] \leftarrow 0 \text{ for some arbitrary } s \in V \end{array} \right.$

while $Q \neq \varnothing$

$|V|$ times

do $u \leftarrow$ EXTRACT-MIN(Q)

$degree(u)$ times

for each $v \in Adj[u]$

do if $v \in Q$ and $w(u, v) < key[v]$

then $key[v] \leftarrow w(u, v)$

$\pi[v] \leftarrow u$

Handshaking Lemma $\Rightarrow$ $\Theta(E)$ implicit DECREASE-KEY's.

ANALYSIS OF PRIM

$\Theta(V)$ total:

$Q \leftarrow V$
$key[v] \leftarrow \infty$ for all $v \in V$
$key[s] \leftarrow 0$ for some arbitrary $s \in V$

$|V|$ times:

while $Q \neq \emptyset$
  do $u \leftarrow$ EXTRACT-MIN(Q)

$degree(u)$ times:

  for each $v \in Adj[u]$
    do if $v \in Q$ and $w(u, v) < key[v]$
      then $key[v] \leftarrow w(u, v)$
        $\pi[v] \leftarrow u$

Handshaking Lemma $\Rightarrow \Theta(E)$ implicit DECREASE-KEY's.

$$\text{Time} = \Theta(V) \cdot T_{\text{EXTRACT-MIN}} + \Theta(E) \cdot T_{\text{DECREASE-KEY}}$$

Analysis of Prim (continued)

Time $= \Theta(V) \cdot T_{\text{EXTRACT-MIN}} + \Theta(E) \cdot T_{\text{DECREASE-KEY}}$

Analysis of Prim (continued)

Time = $\Theta(V) \cdot T_{\text{EXTRACT-MIN}} + \Theta(E) \cdot T_{\text{DECREASE-KEY}}$

Q	$T_{\text{EXTRACT-MIN}}$	$T_{\text{DECREASE-KEY}}$	Total

Analysis of Prim (continued)

$$\text{Time} = \Theta(V)\cdot T_{\text{EXTRACT-MIN}} + \Theta(E)\cdot T_{\text{DECREASE-KEY}}$$

Q	$T_{\text{EXTRACT-MIN}}$	$T_{\text{DECREASE-KEY}}$	Total
array	$O(V)$	$O(1)$	$O(V^2)$

ANALYSIS OF PRIM (CONTINUED)

$$\text{Time} = \Theta(V)\cdot T_{\text{EXTRACT-MIN}} + \Theta(E)\cdot T_{\text{DECREASE-KEY}}$$

Q	$T_{\text{EXTRACT-MIN}}$	$T_{\text{DECREASE-KEY}}$	Total
array	$O(V)$	$O(1)$	$O(V^2)$
binary heap	$O(\lg V)$	$O(\lg V)$	$O(E \lg V)$

Analysis of Prim (continued)

Time = $\Theta(V) \cdot T_{\text{EXTRACT-MIN}} + \Theta(E) \cdot T_{\text{DECREASE-KEY}}$

Q	$T_{\text{EXTRACT-MIN}}$	$T_{\text{DECREASE-KEY}}$	Total
array	$O(V)$	$O(1)$	$O(V^2)$
binary heap	$O(\lg V)$	$O(\lg V)$	$O(E \lg V)$
Fibonacci heap	$O(\lg V)$ amortized	$O(1)$ amortized	$O(E + V \lg V)$ worst case

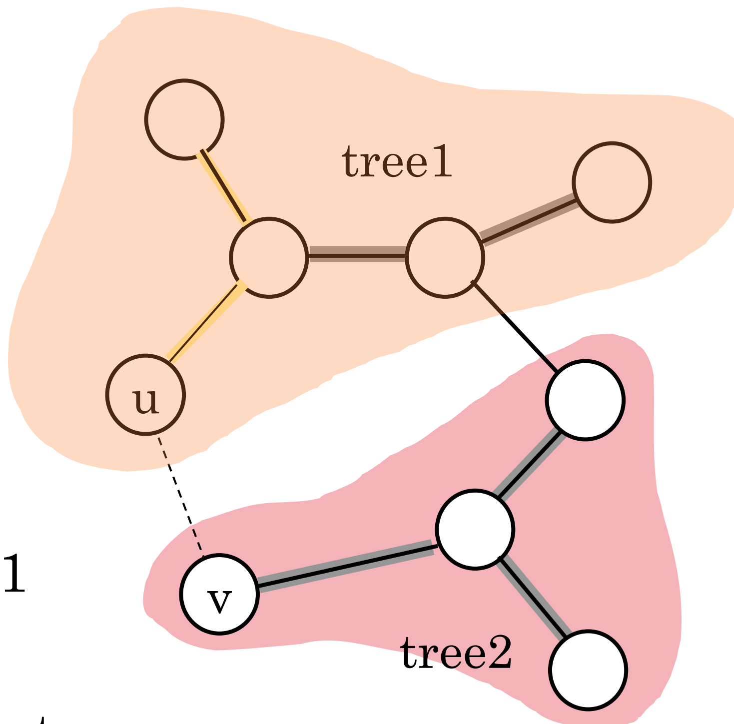
MST ALGORITHMS

Kruskal's algorithm (see CLRS):

- Uses the ***disjoint-set data structure*** (see CLRS, Ch. 21).
- Running time $= O(E \lg V)$.

KRUSKAL'S ALGORITHM

- How is it different from Prim's algorithm?
  - Prim's algorithm grows one tree all the time
  - Kruskal's algorithm grows multiple trees (i.e., a forest) at the same time.
  - Trees are merged together using **safe** edges
  - Since an MST has exactly |V| - 1 edges, after |V| - 1 merges, we would have only one component



Kruskal's Algorithm

- Kruskal's algorithm creates a forest of trees
- Initially forest consists of N single node trees (and no edges)
- Sorts the edges by weight and goes through the edges from least weight to greatest weight
- At each step one edge (with least weight) is added so that it joins two trees together
  - As long as the addition does not create a cycle

Kruskal's Algorithm

The halting conditions are as follows:

1. When `|V| - 1` edges have been added
   - In this case we have a minimum spanning tree

2. We have gone through all edges
   - A forest of minimum spanning trees on all connected sub-graphs

Kruskal's Algorithm

1. The forest is constructed – with each node in a separate tree

2. The edges are placed in a priority queue

3. Until we've added `n-1` edges (assumption: connected graph)
    1. Extract the cheapest edge from the queue

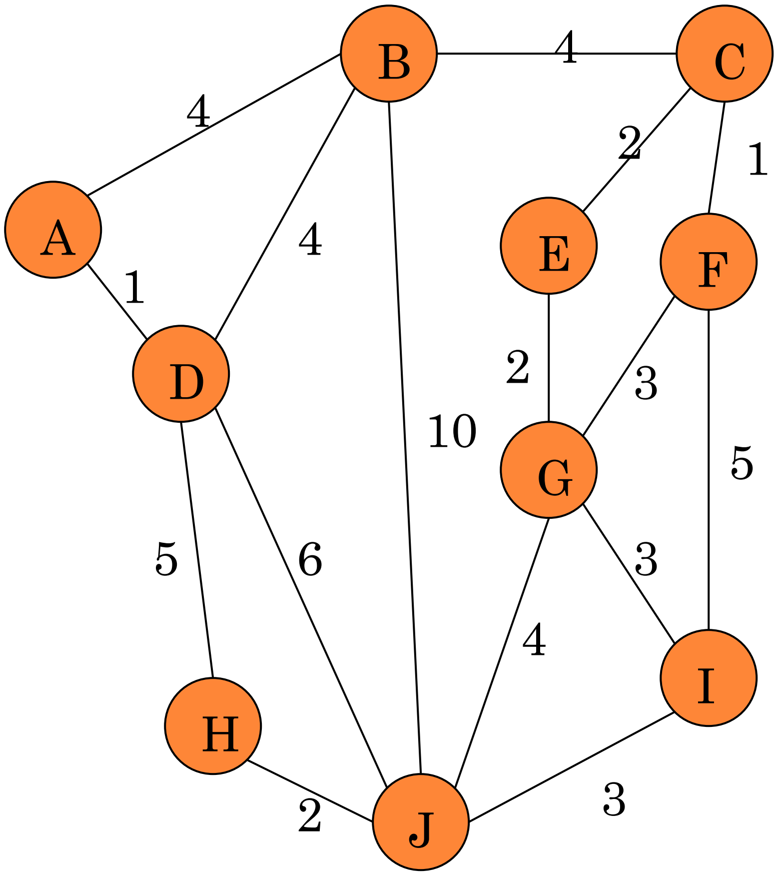
    2. If it forms a cycle, reject it

    3. Else add it to the forest. Adding it to the forest will join two trees

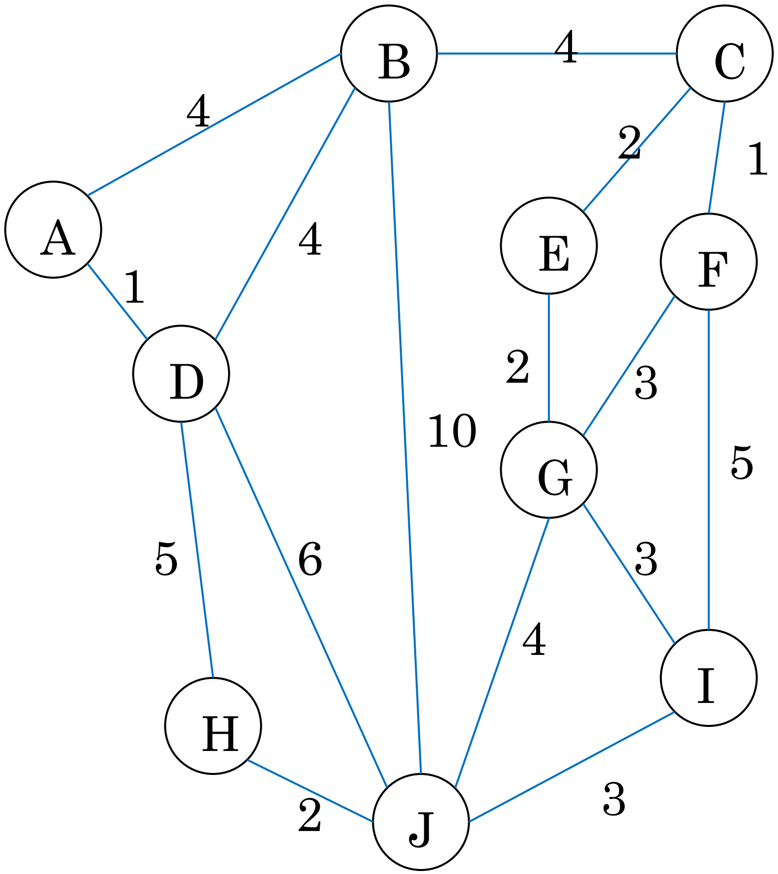
- Every step will have joined two trees in the forest together, so that at the end, there will only be one tree

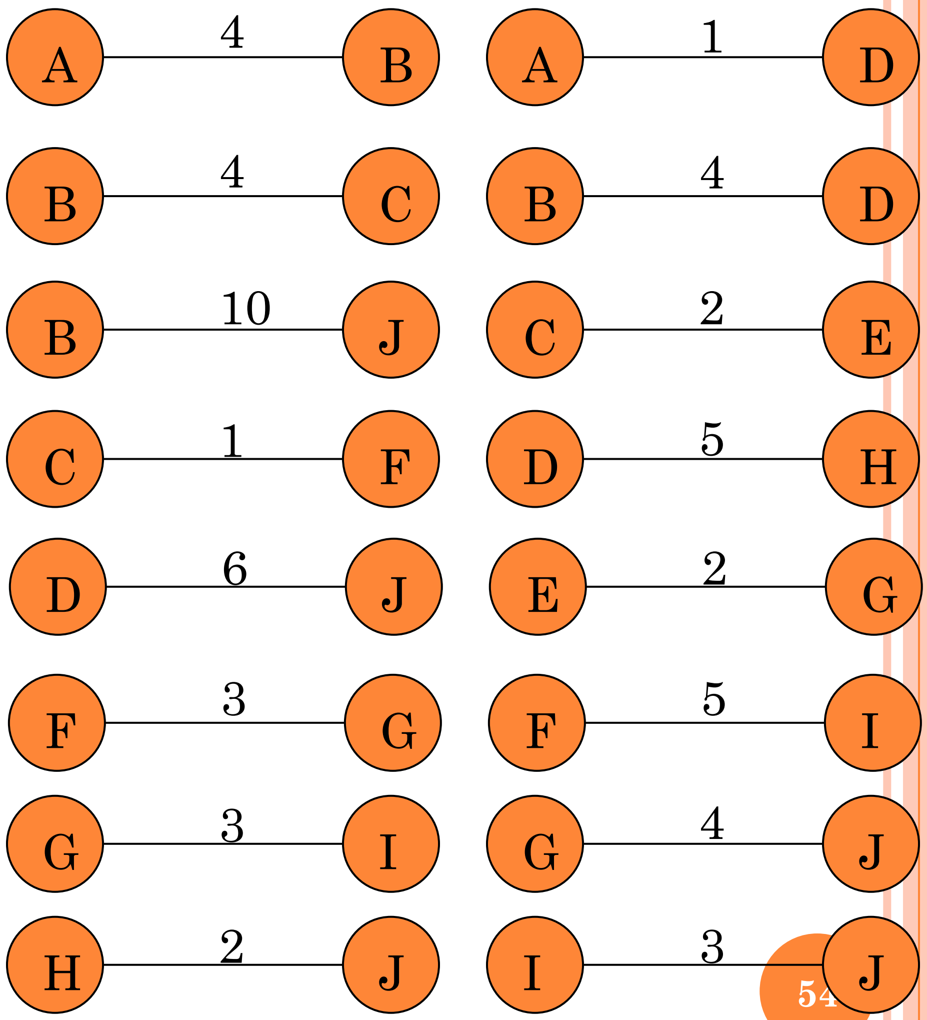
Kruskal's Algorithm – Example

- Complete Graph



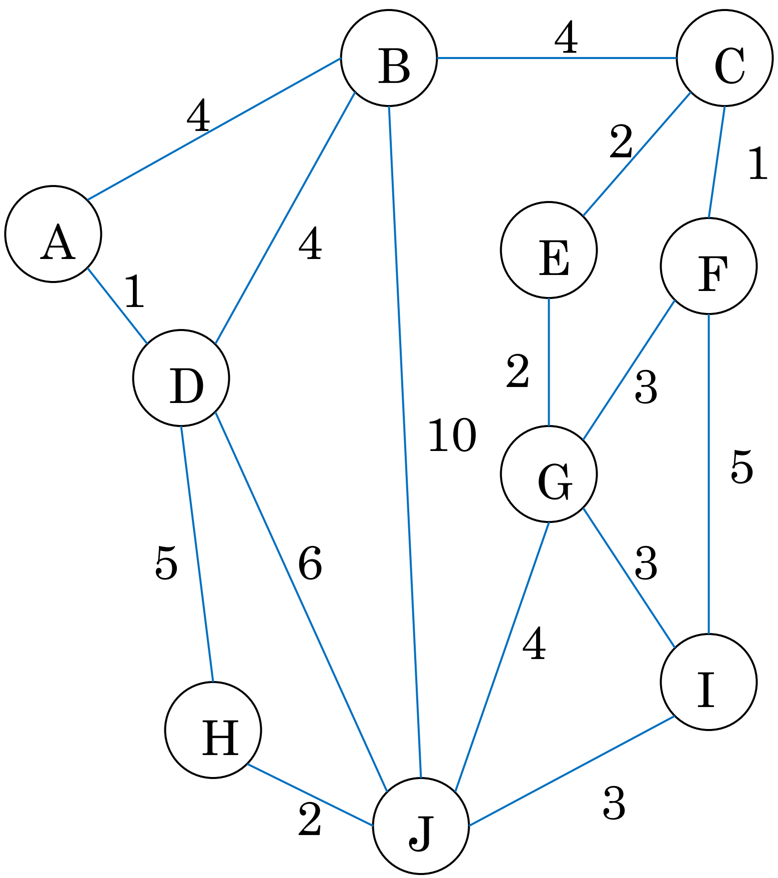
Kruskal's Algorithm – Example

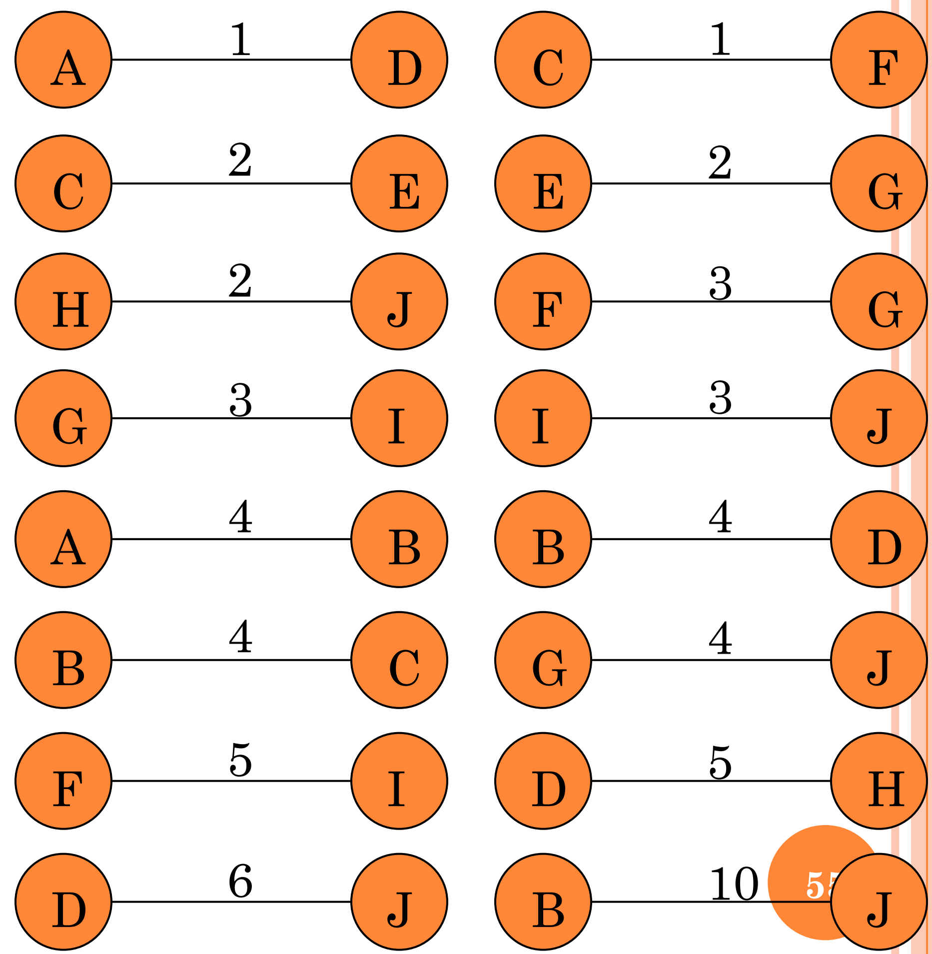




Kruskal's Algorithm – Example

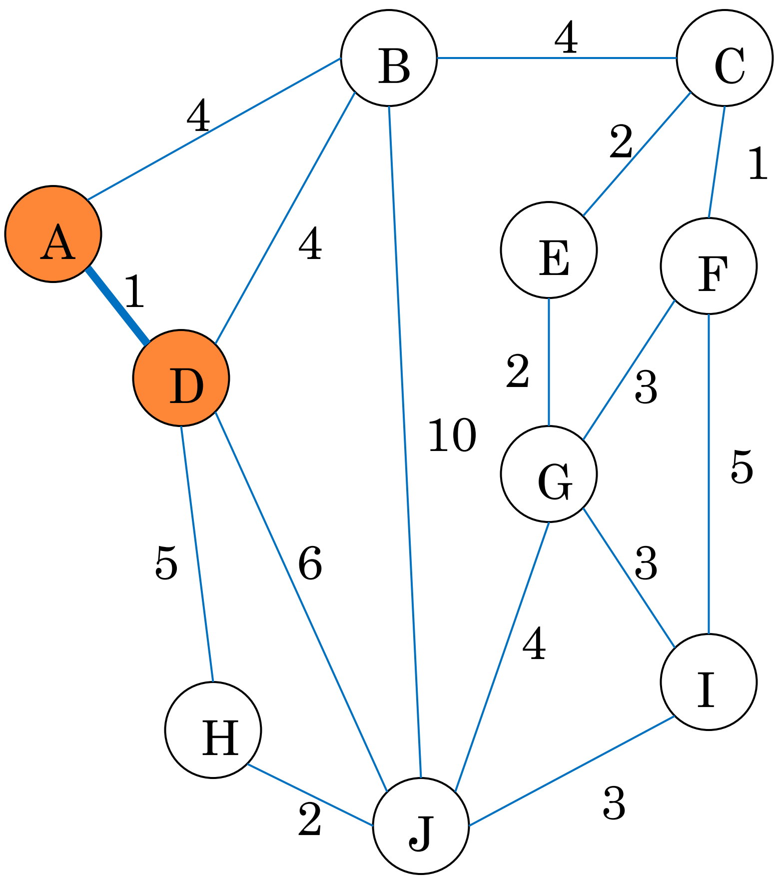
- Sort Edges: In reality priority queue is used

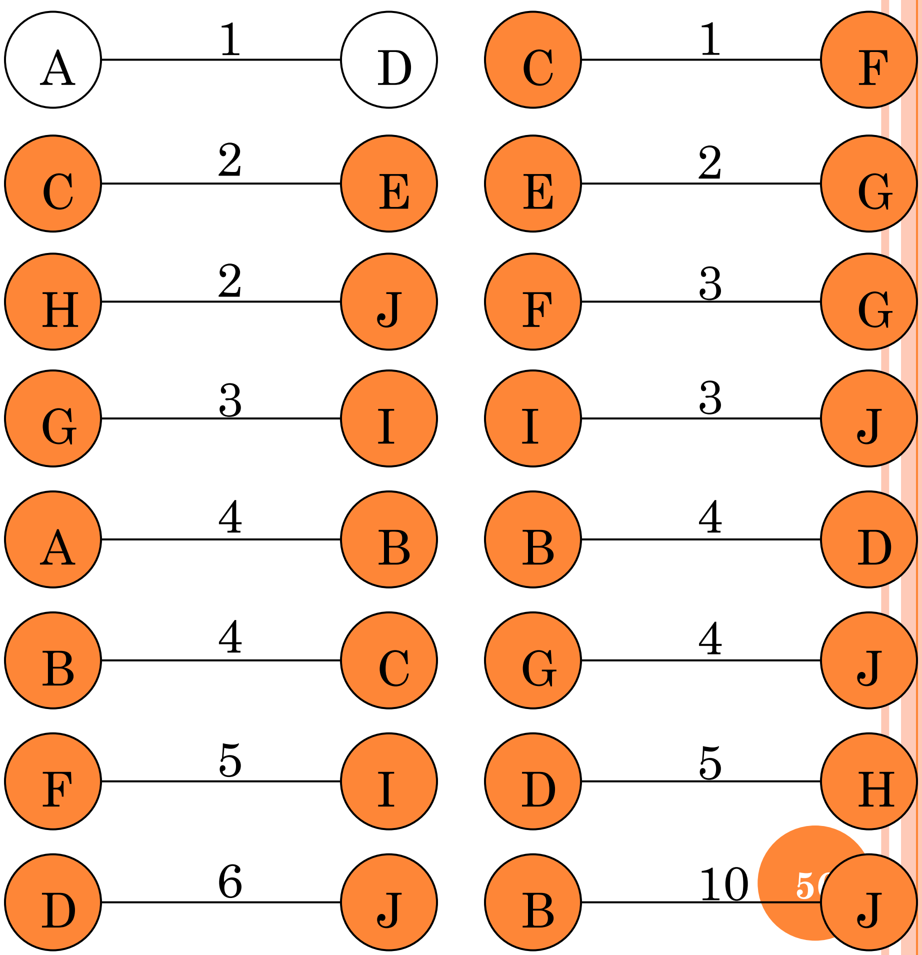




Kruskal's Algorithm – Example

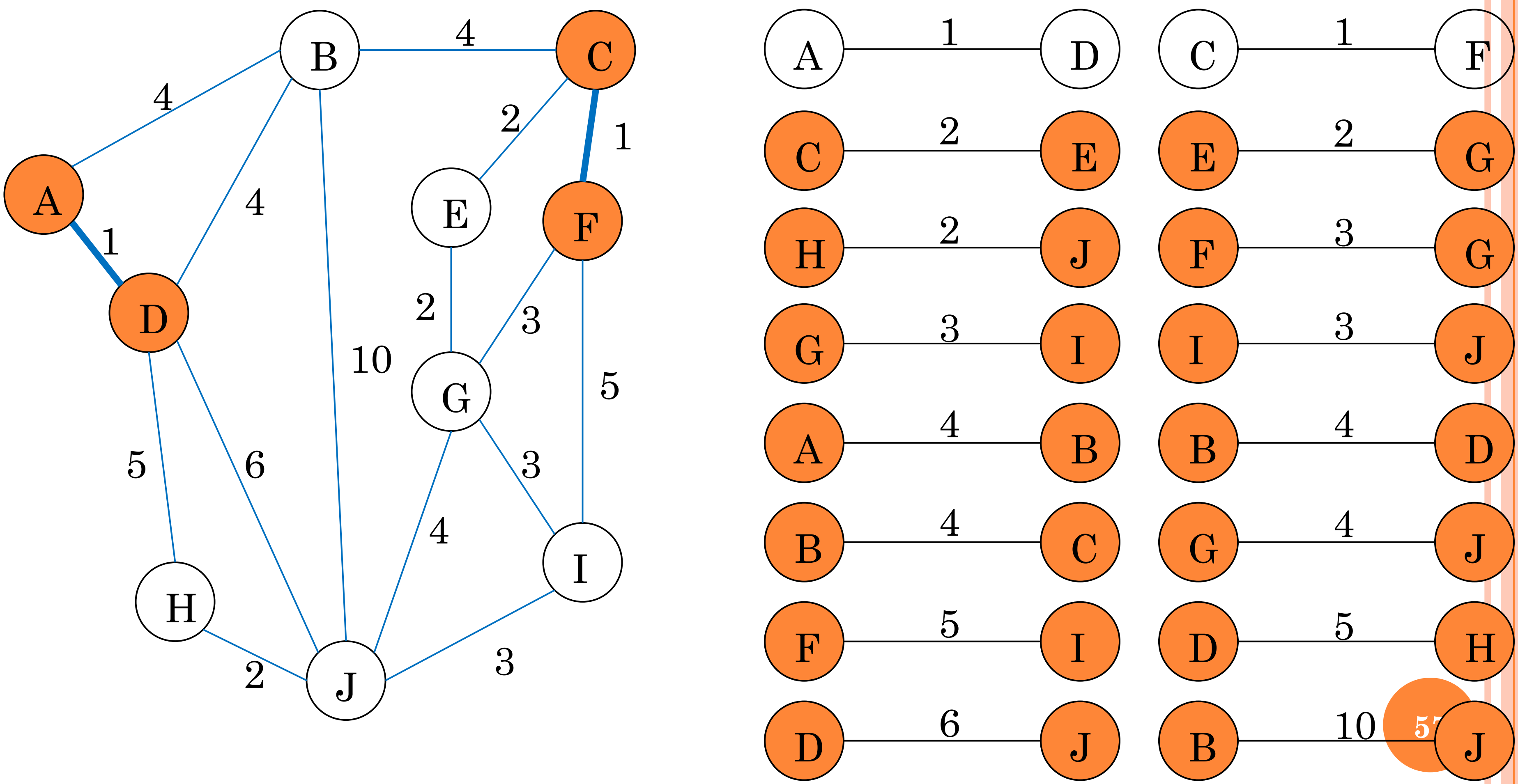
- Add Edge





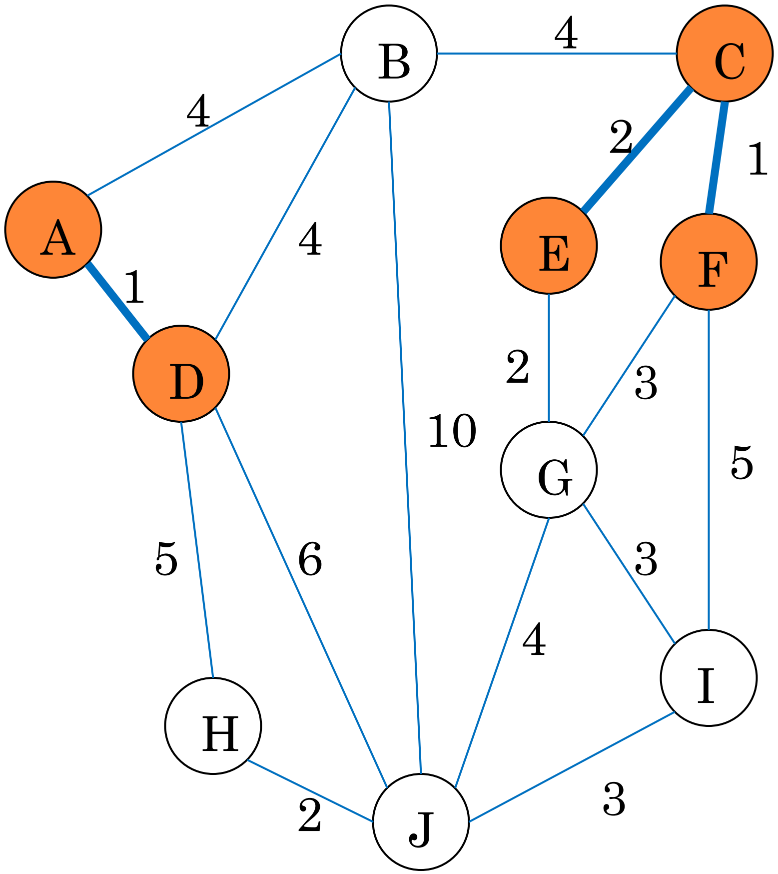
Kruskal's Algorithm – Example

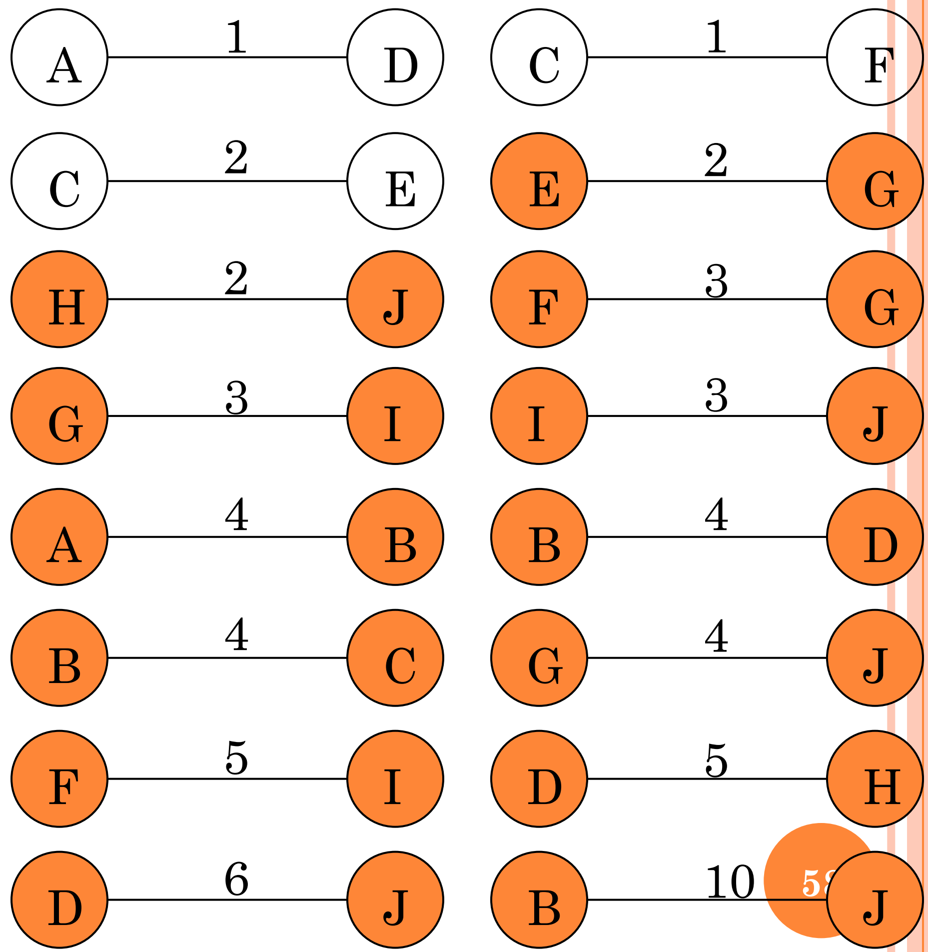
- Add Edge



Kruskal's Algorithm – Example

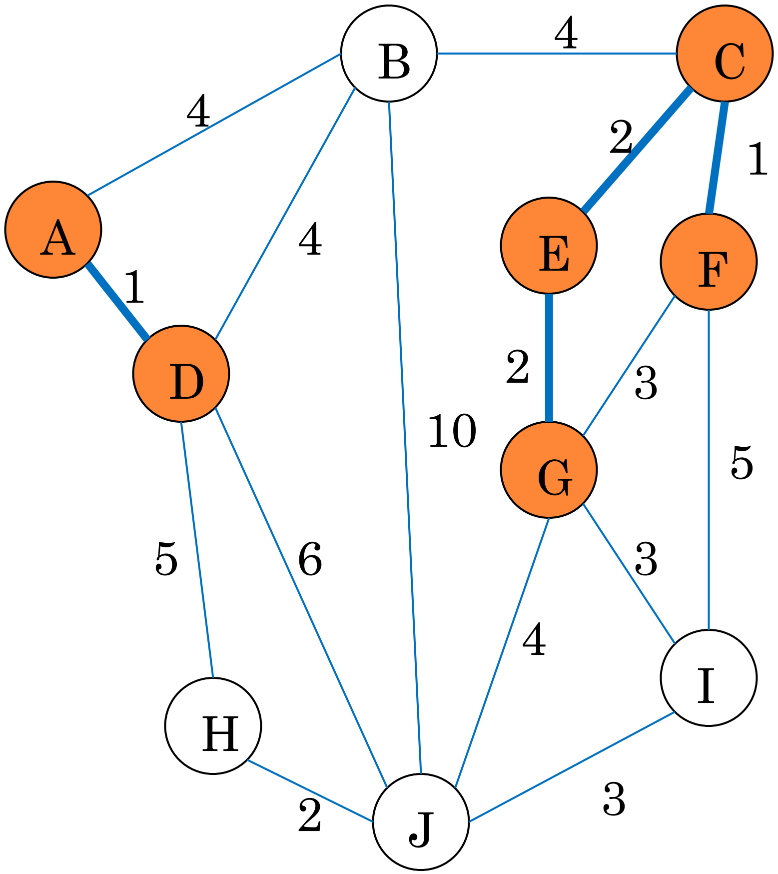
- Add Edge

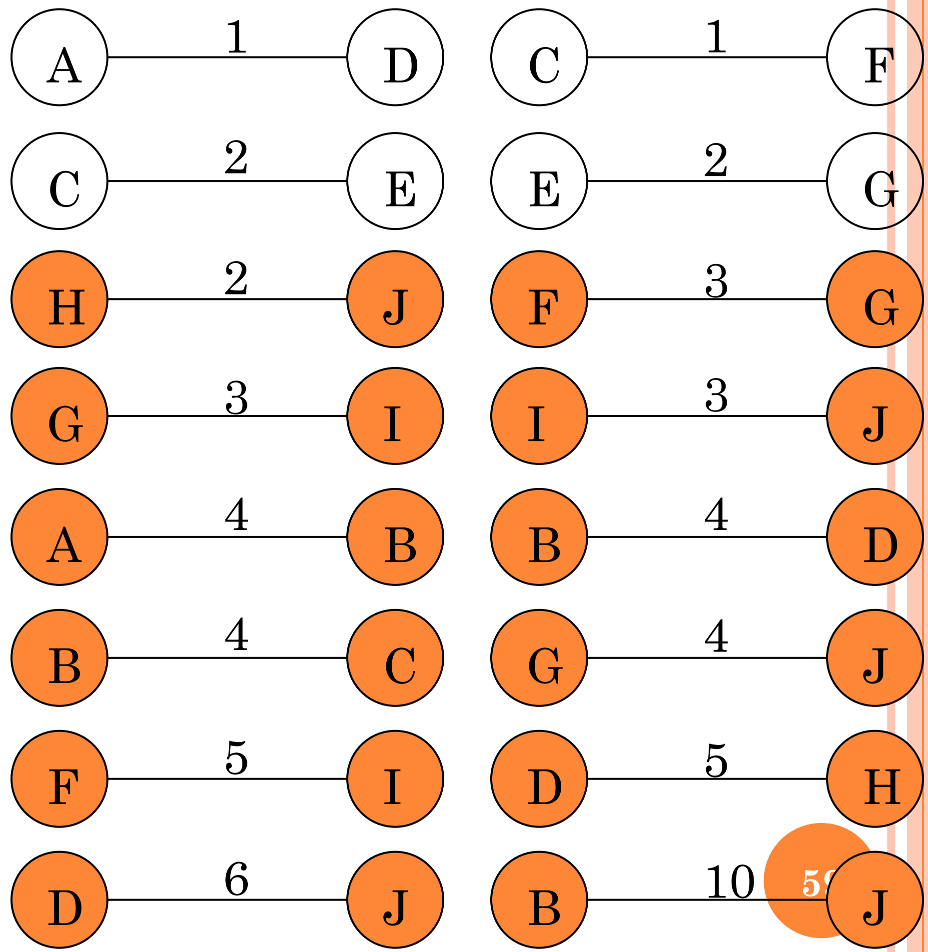




Kruskal's Algorithm – Example

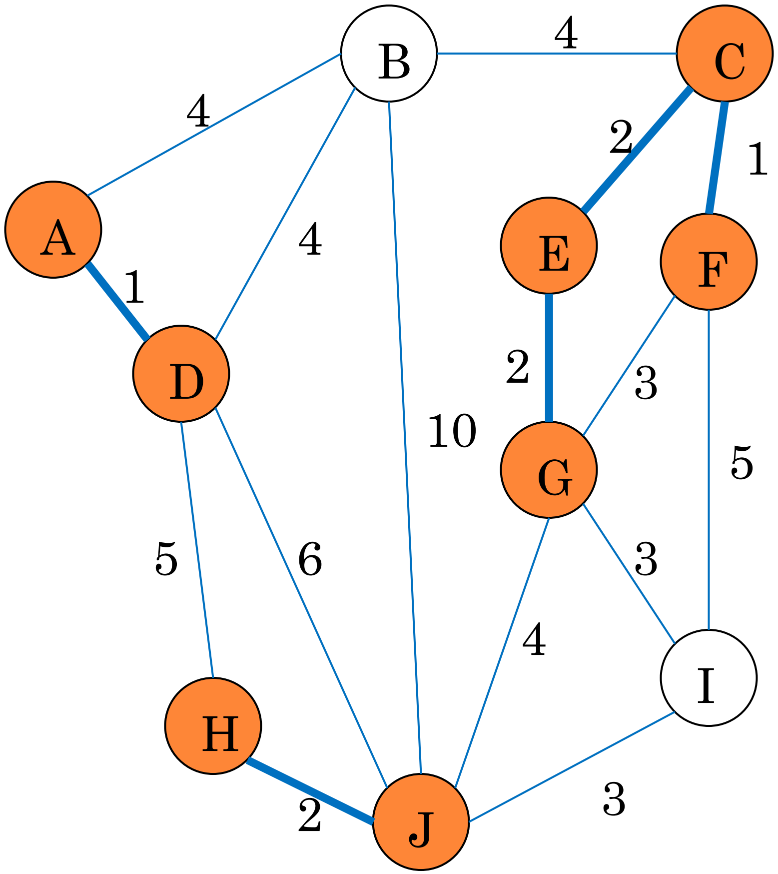
- Add Edge

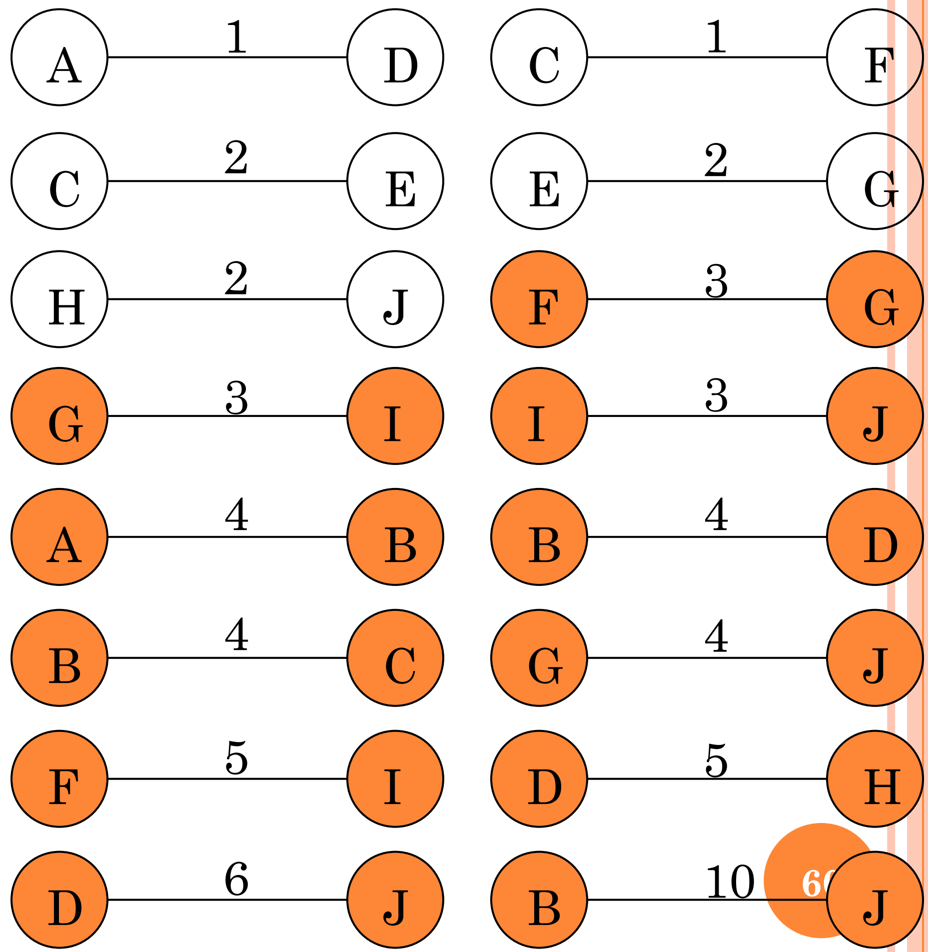




Kruskal's Algorithm – Example

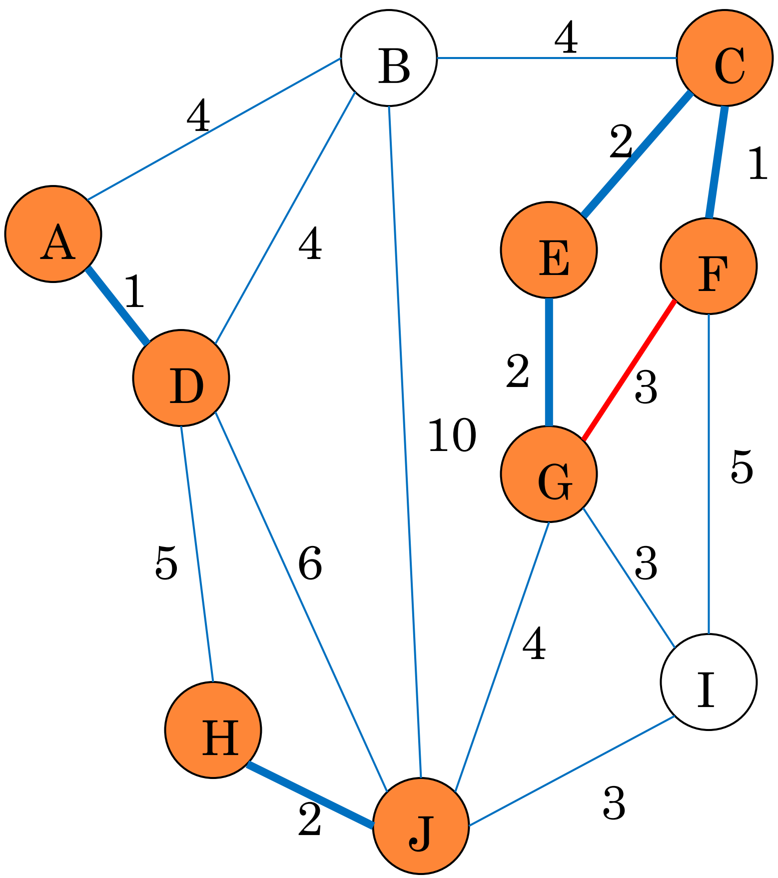
- Add Edge

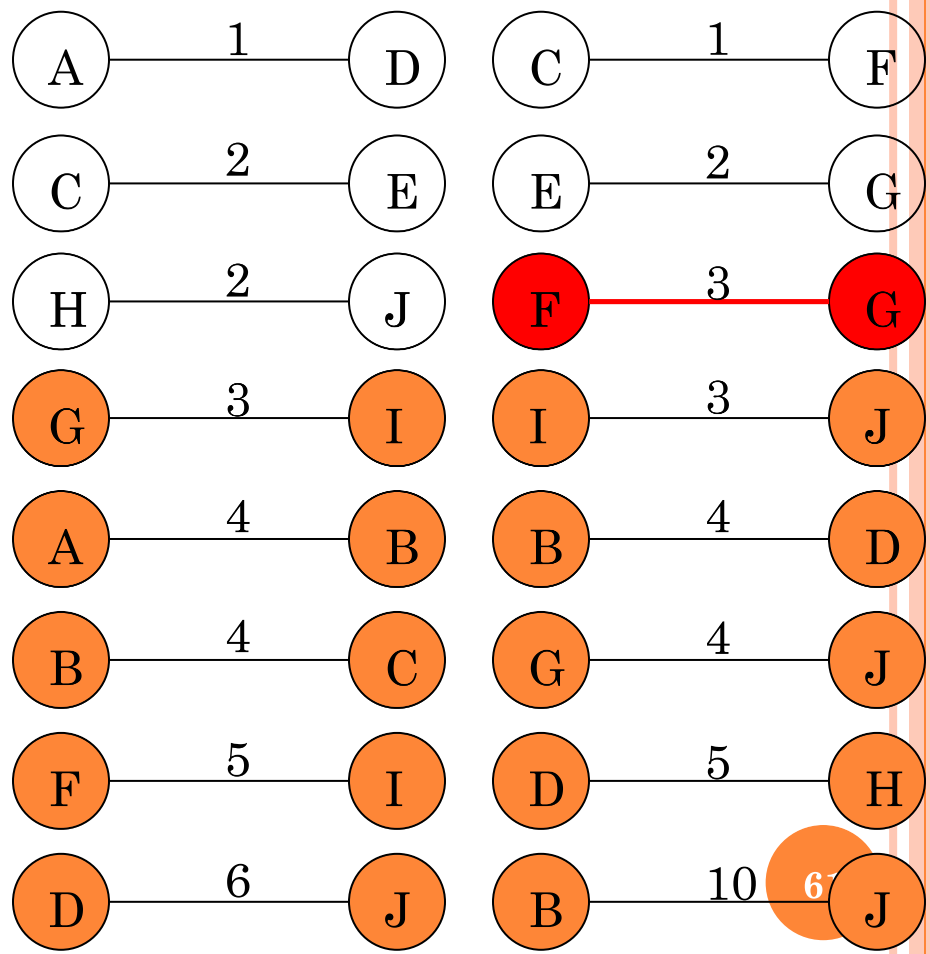




Kruskal's Algorithm – Example

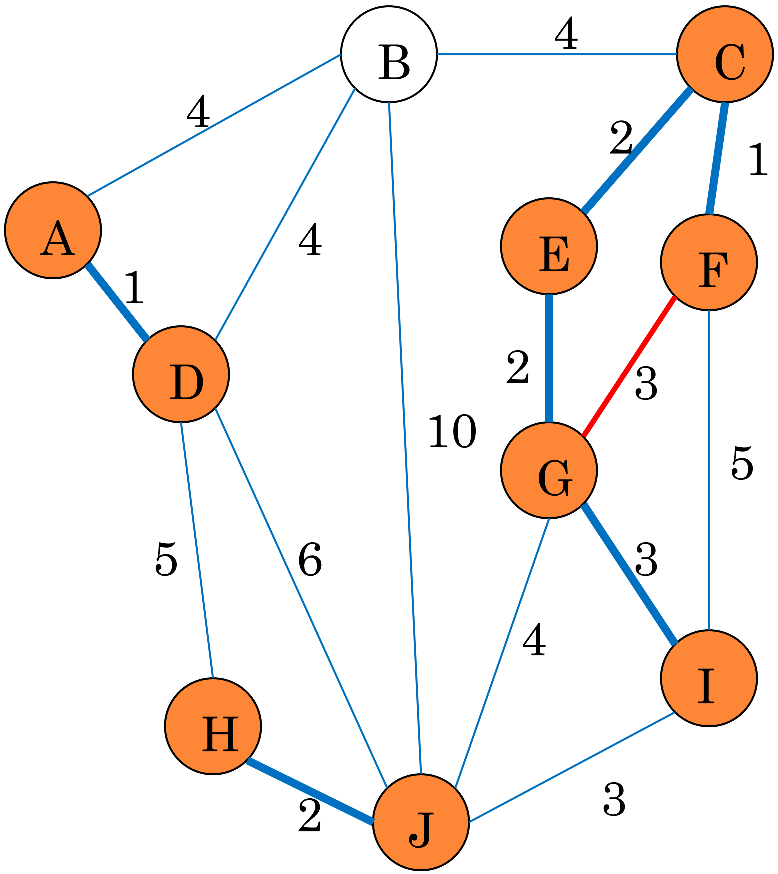
- Add Edge

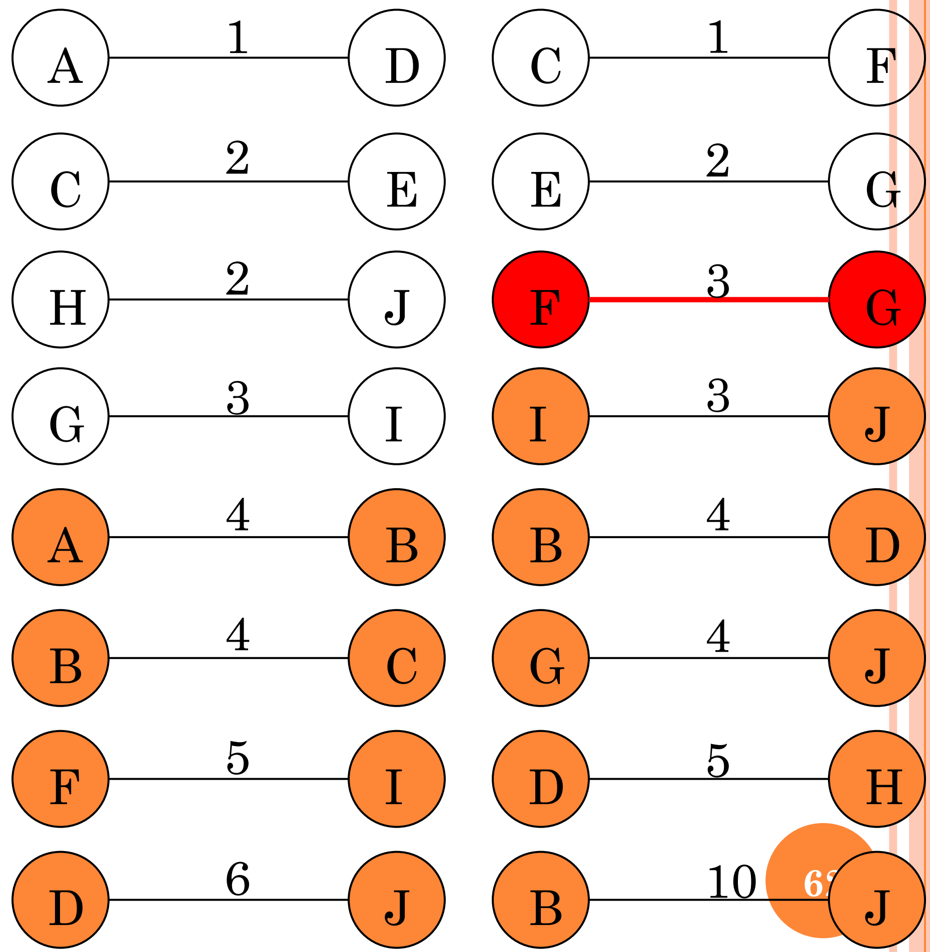




Kruskal's Algorithm – Example

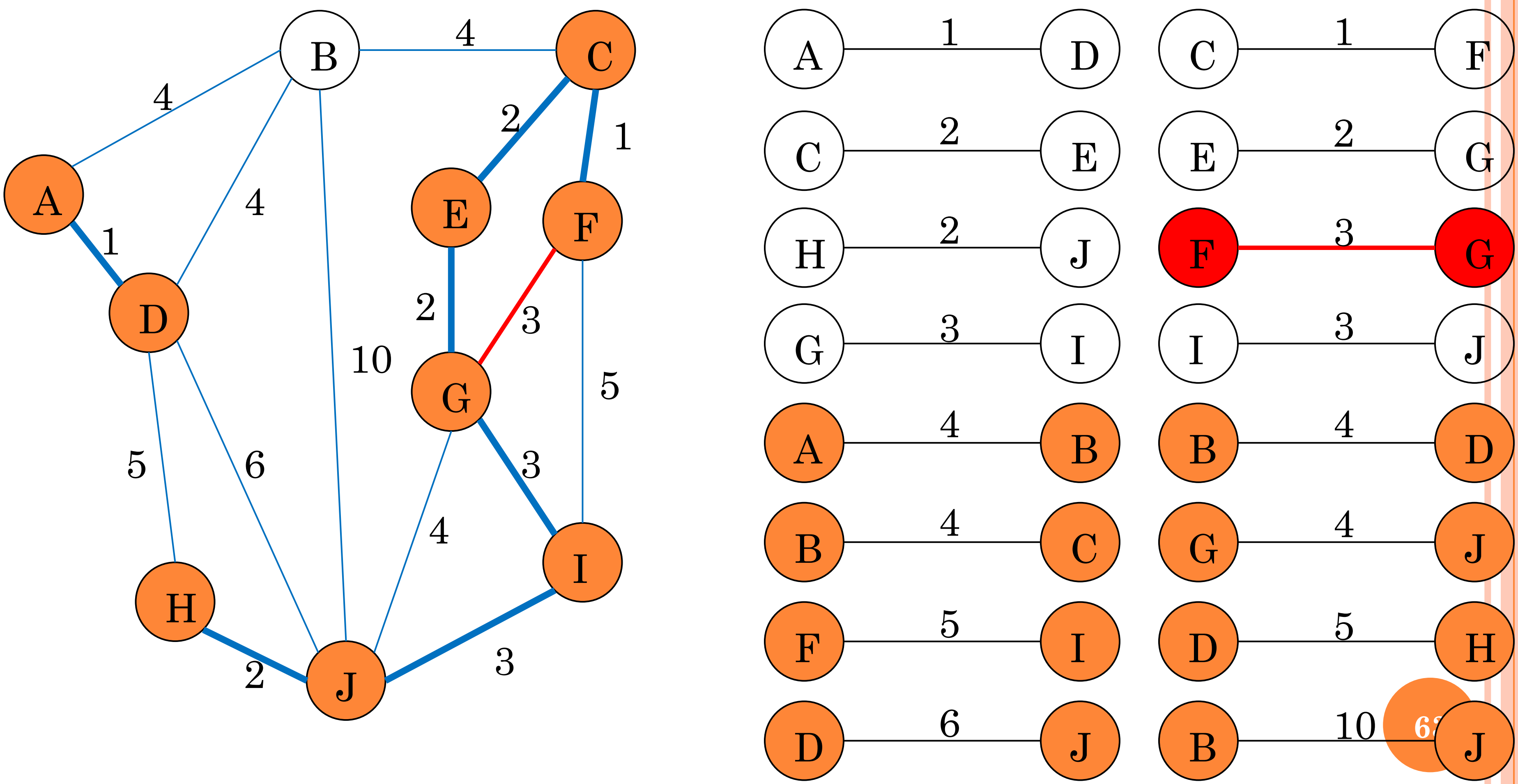
- Add Edge





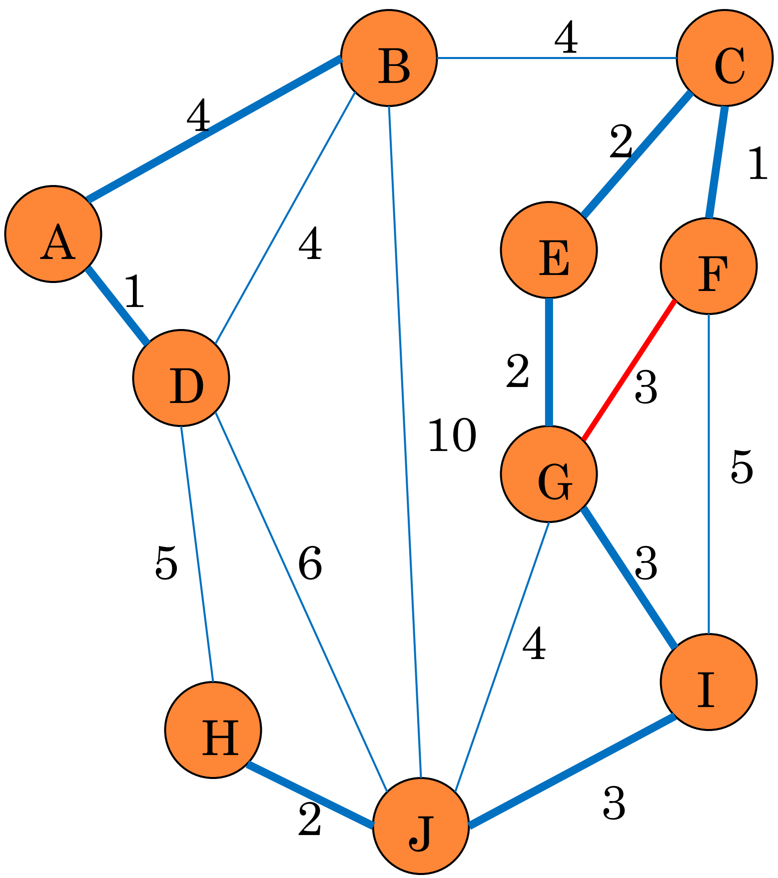
Kruskal's Algorithm – Example

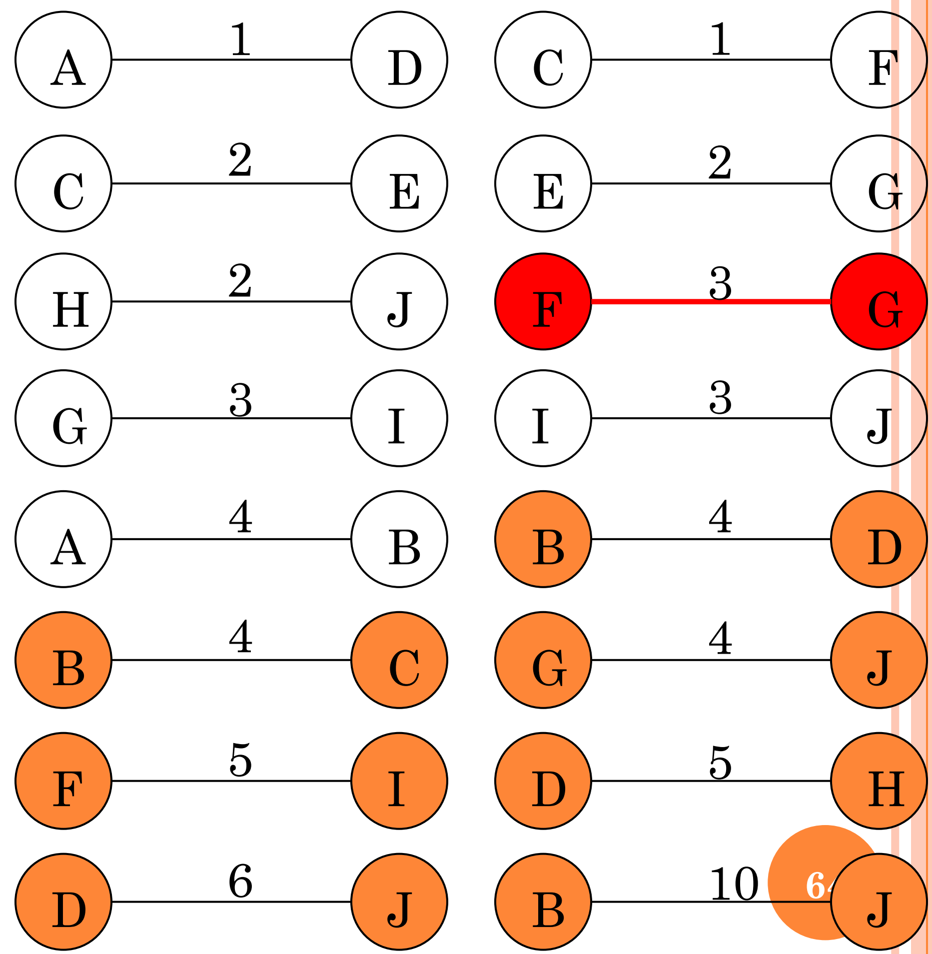
- Add Edge



Kruskal's Algorithm – Example

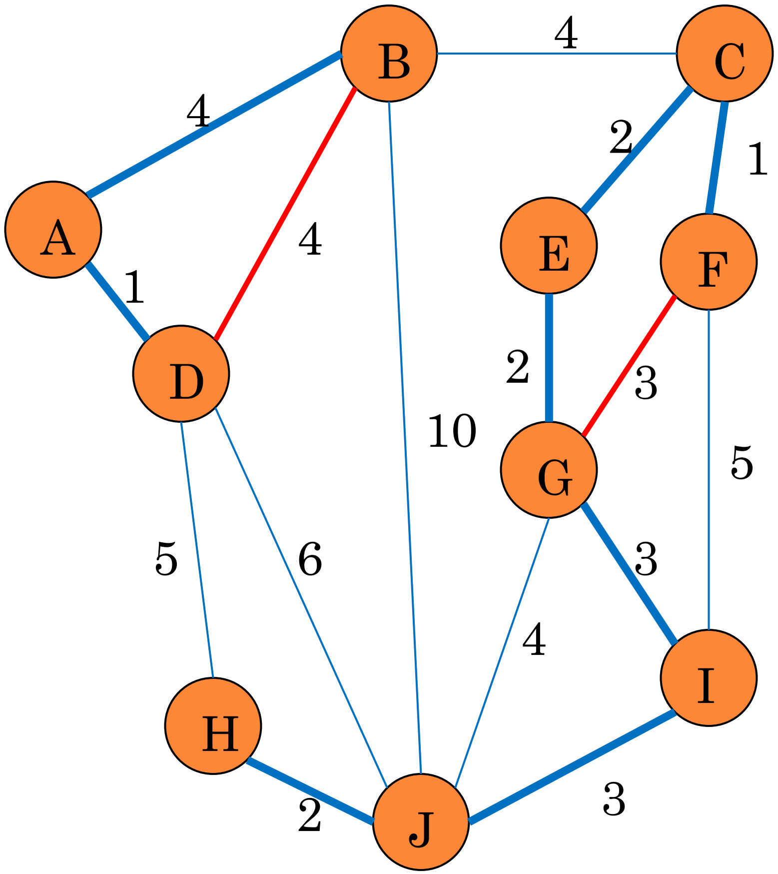
- Add Edge

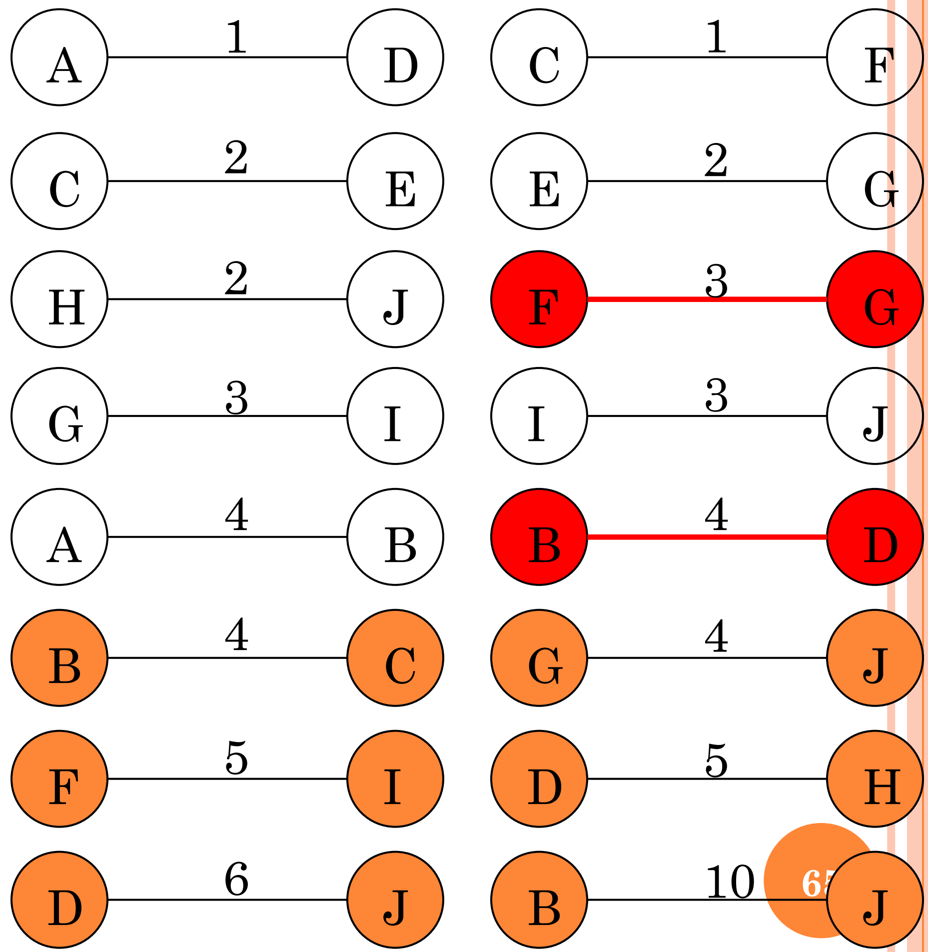




Kruskal's Algorithm – Example

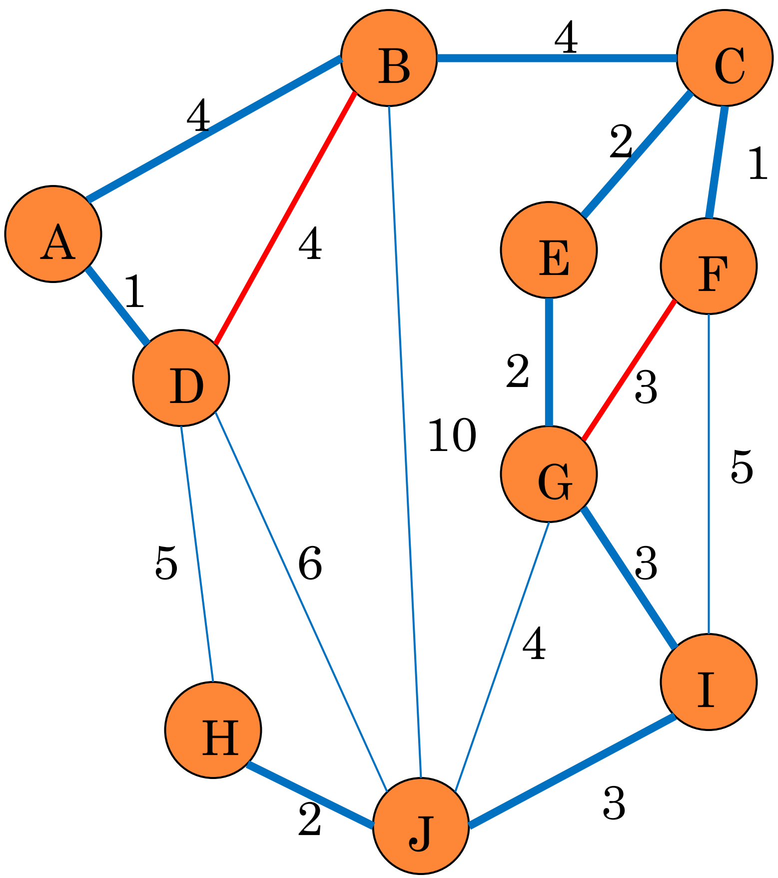
- Add Edge

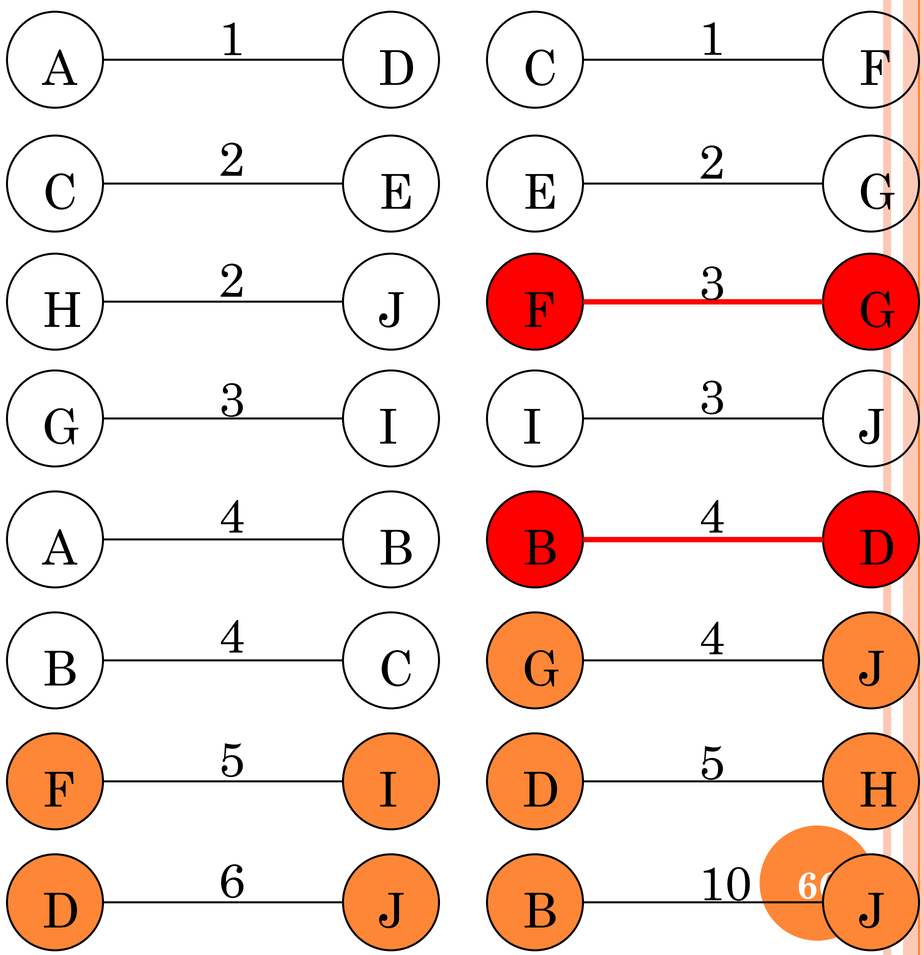




Kruskal's Algorithm – Example

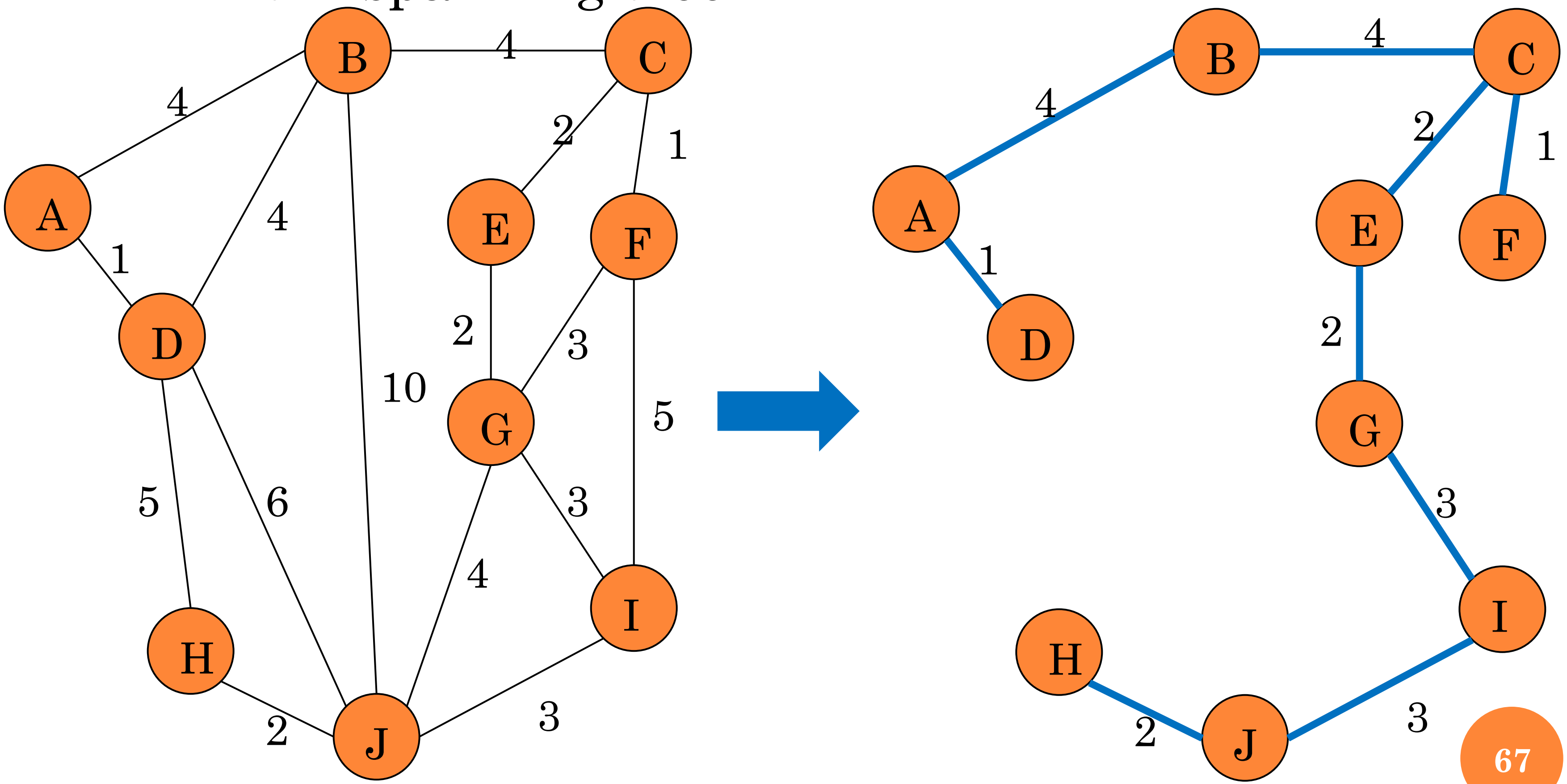
- Add Edge





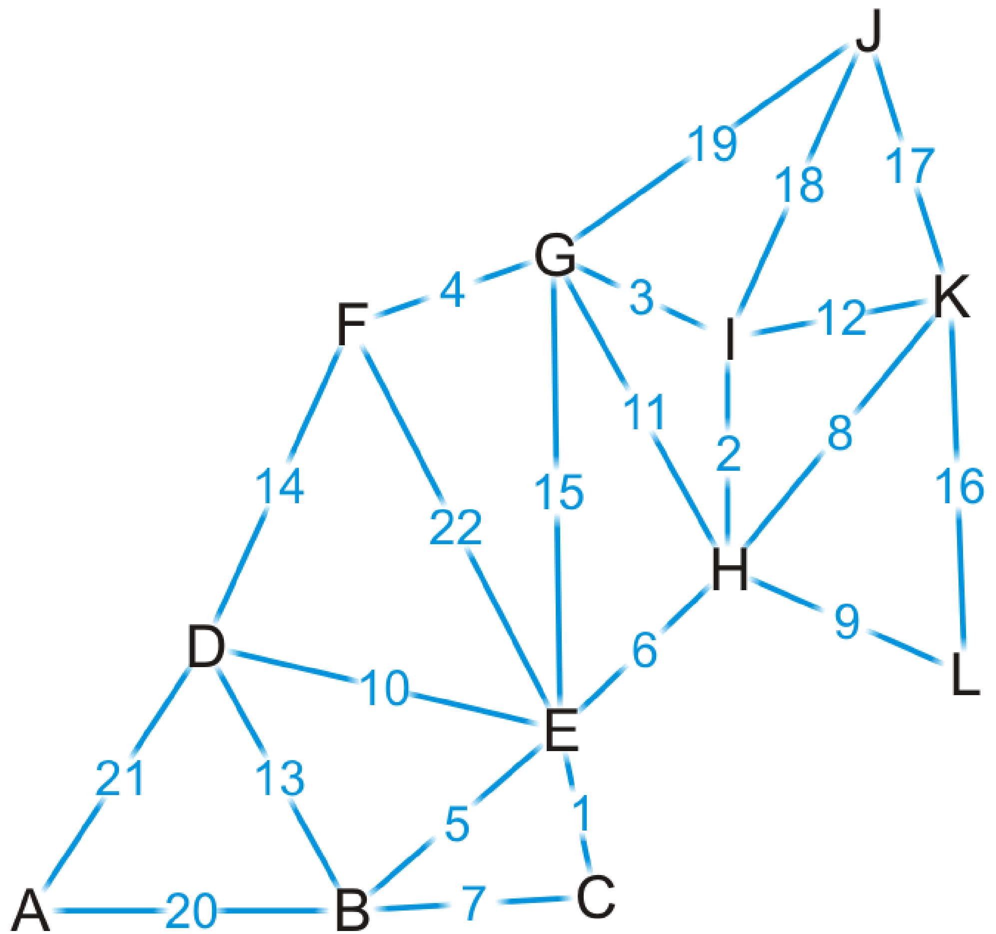
Kruskal's Algorithm – Example

- Minimum spanning tree

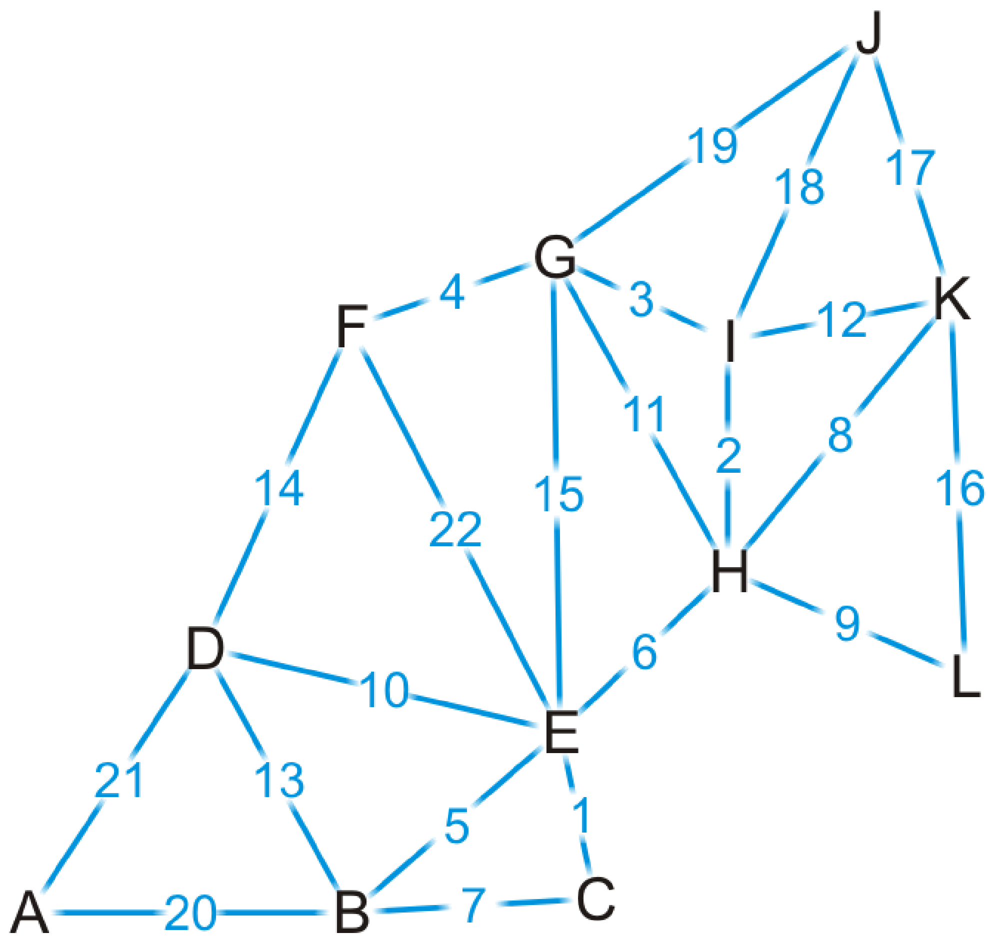


Kruskal's Algorithm – Example

- Complete graph

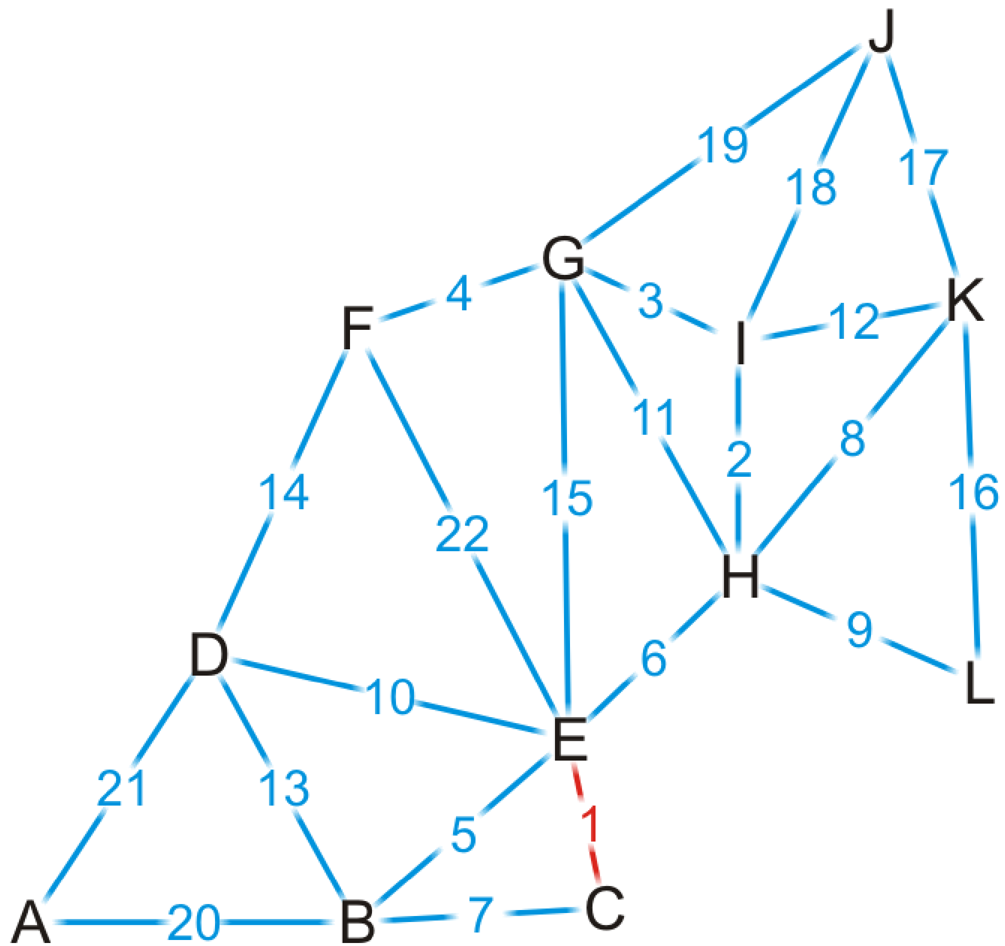


Kruskal's Algorithm – Example



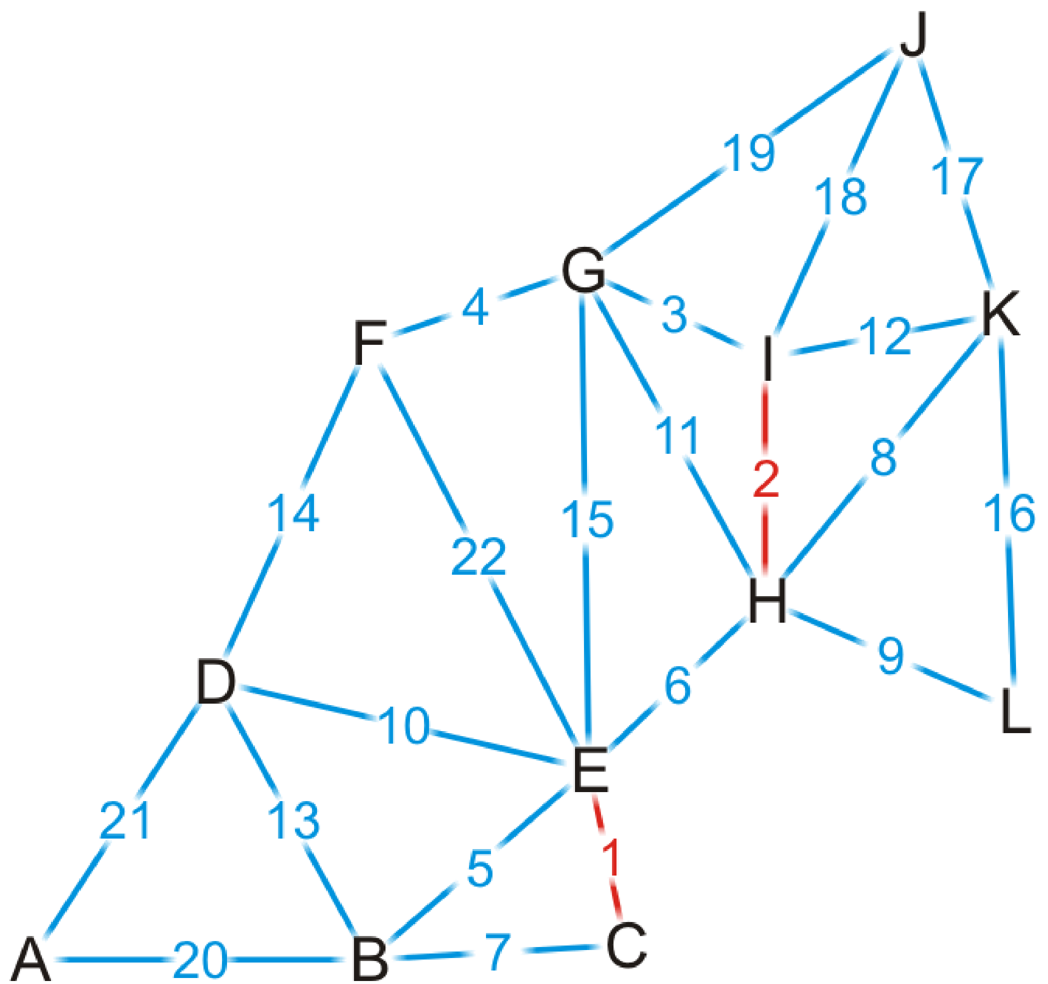
{C, E}
{H, I}
{G, I}
{F, G}
{B, E}
{E, H}
{B, C}
{H, K}
{H, L}
{D, E}
{G, H}
{I, K}
{B, D}
{D, F}
{E, G}
{K, L}
{J, K}
{J, I}
{J, G}
{A, B}
{A, D}
{E, F}

Kruskal's Algorithm – Example



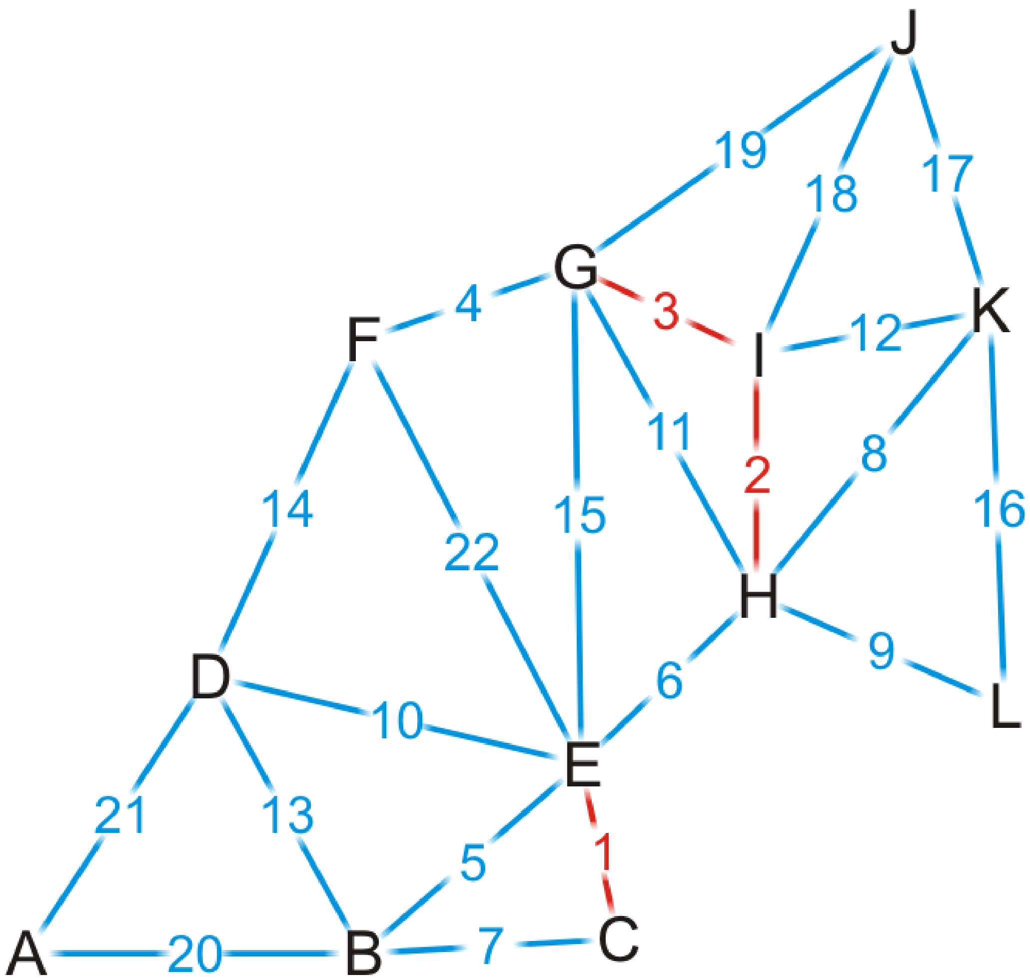
{C, E}
{H, I}
{G, I}
{F, G}
{B, E}
{E, H}
{B, C}
{H, K}
{H, L}
{D, E}
{G, H}
{I, K}
{B, D}
{D, F}
{E, G}
{K, L}
{J, K}
{J, I}
{J, G}
{A, B}
{A, D}
{E, F}

KRUSKAL'S ALGORITHM – EXAMPLE



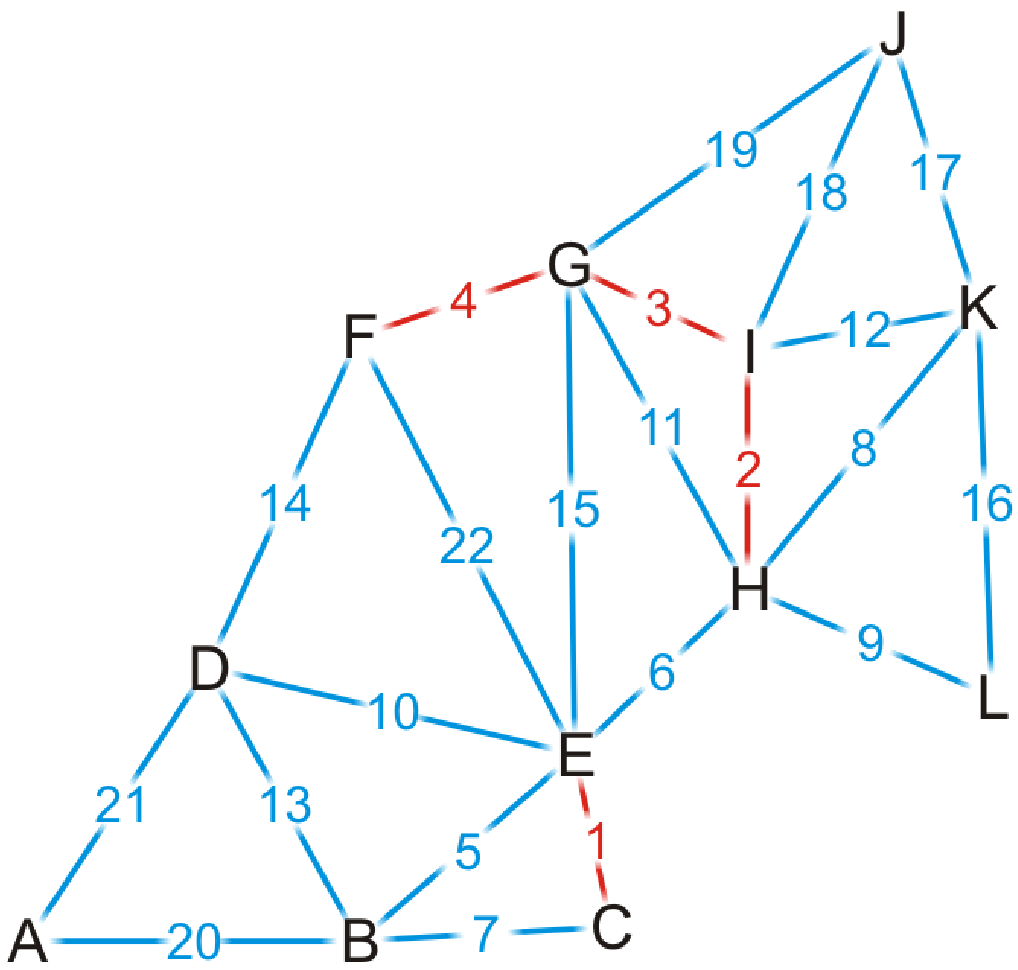
{C, E}
{H, I}
{G, I}
{F, G}
{B, E}
{E, H}
{B, C}
{H, K}
{H, L}
{D, E}
{G, H}
{I, K}
{B, D}
{D, F}
{E, G}
{K, L}
{J, K}
{J, I}
{J, G}
{A, B}
{A, D}
{E, F}

KRUSKAL'S ALGORITHM – EXAMPLE



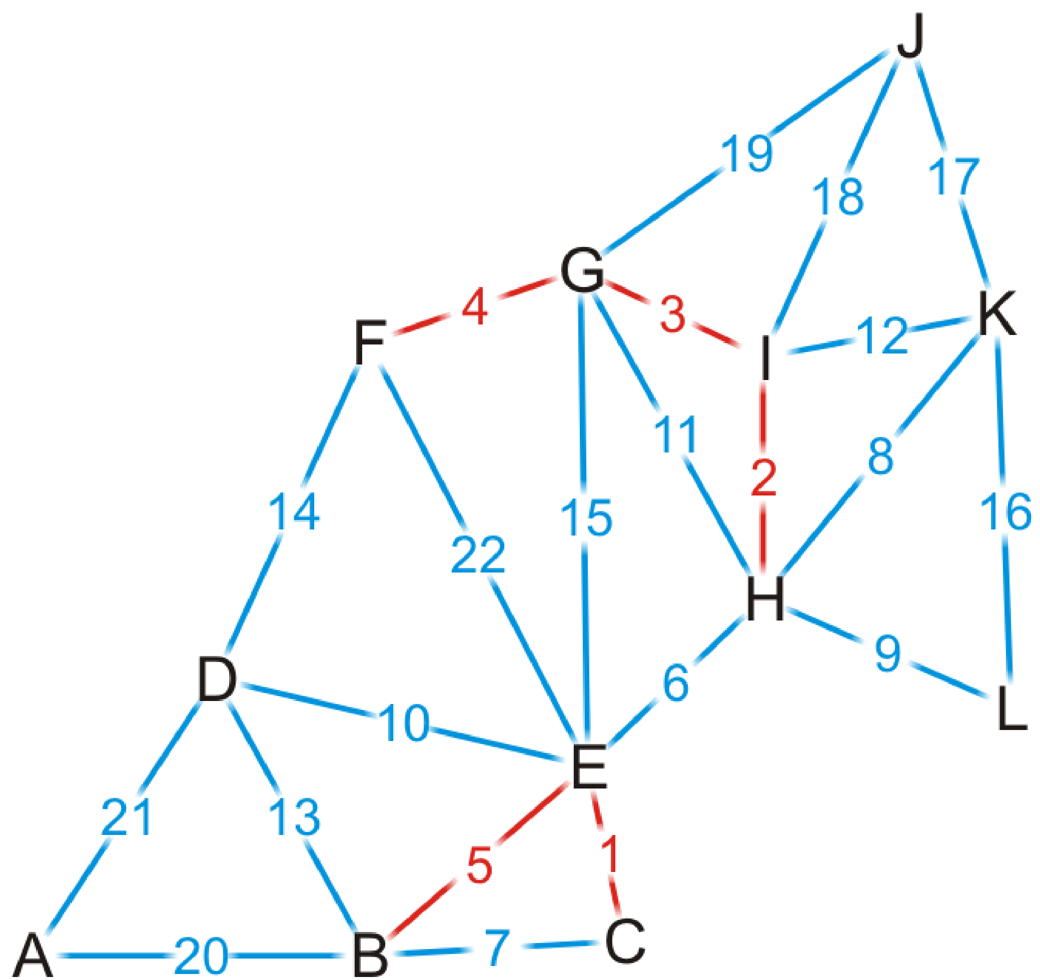
{C, E}
{H, I}
{G, I}
{F, G}
{B, E}
{E, H}
{B, C}
{H, K}
{H, L}
{D, E}
{G, H}
{I, K}
{B, D}
{D, F}
{E, G}
{K, L}
{J, K}
{J, I}
{J, G}
{A, B}
{A, D}
{E, F}

Kruskal's Algorithm – Example



{C, E}
{H, I}
{G, I}
{F, G}
{B, E}
{E, H}
{B, C}
{H, K}
{H, L}
{D, E}
{G, H}
{I, K}
{B, D}
{D, F}
{E, G}
{K, L}
{J, K}
{J, I}
{J, G}
{A, B}
{A, D}
{E, F}

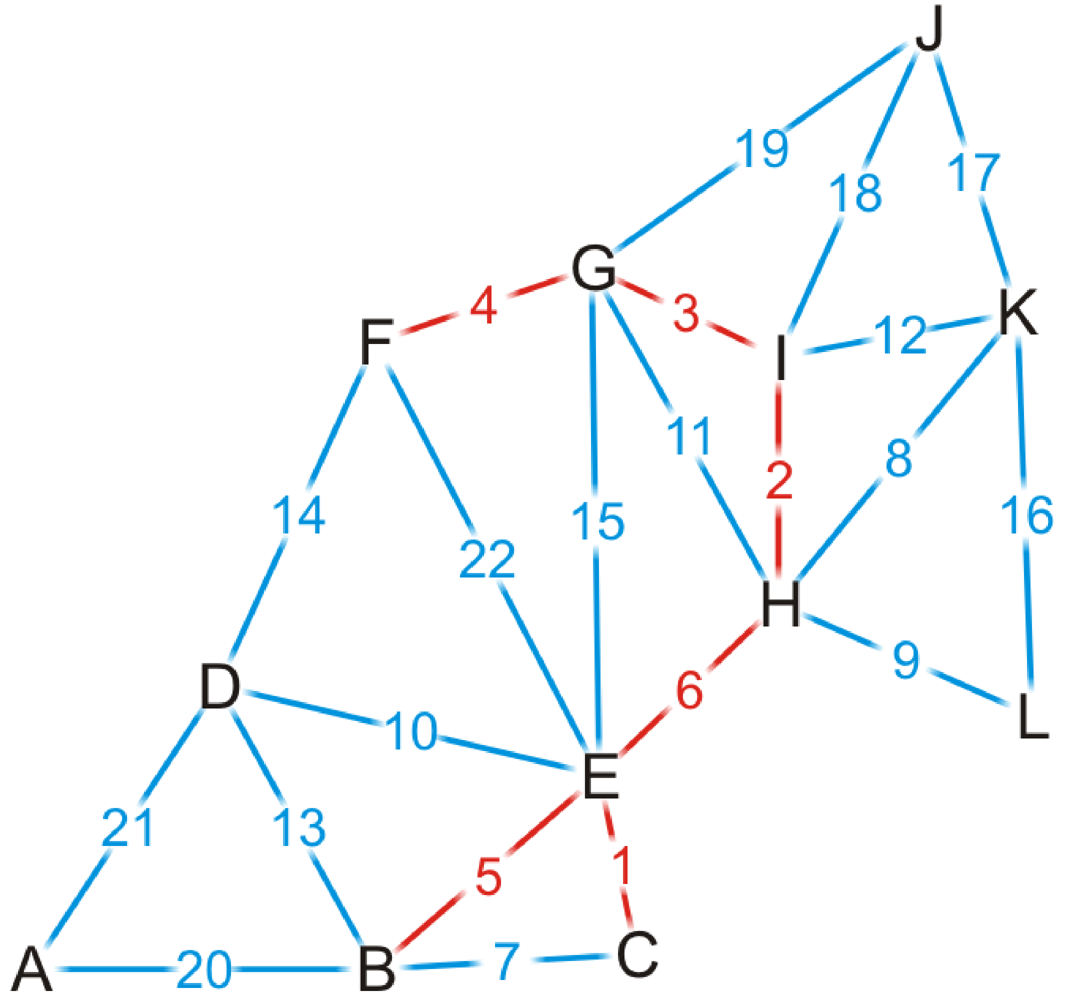
Kruskal's Algorithm – Example



{C, E}
{H, I}
{G, I}
{F, G}
{B, E}
{E, H}
{B, C}
{H, K}
{H, L}
{D, E}
{G, H}
{I, K}
{B, D}
{D, F}
{E, G}
{K, L}
{J, K}
{J, I}
{J, G}
{A, B}
{A, D}
{E, F}

Kruskal's Algorithm – Example

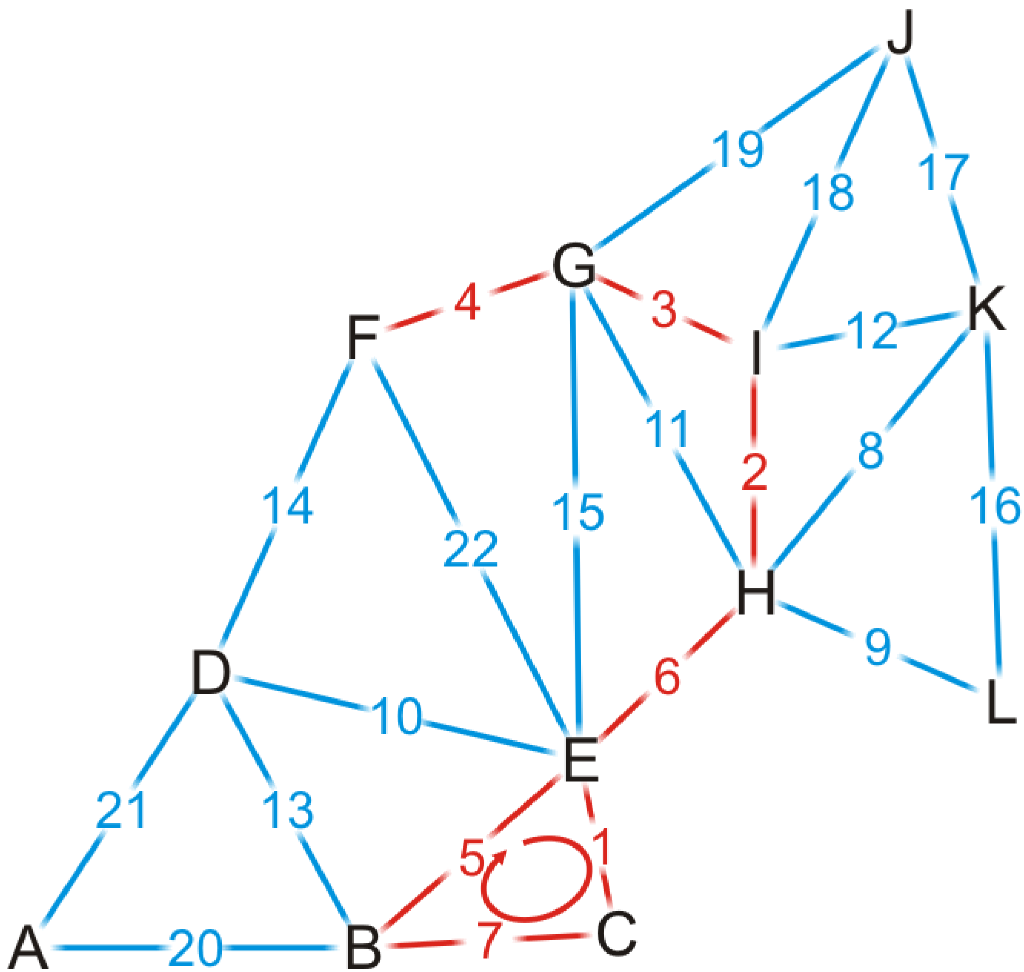
- We add edge {E, H}
  - This coalesces the two spanning sub-trees into one



{C, E}
{H, I}
{G, I}
{F, G}
{B, E}
{E, H}
{B, C}
{H, K}
{H, L}
{D, E}
{G, H}
{I, K}
{B, D}
{D, F}
{E, G}
{K, L}
{J, K}
{J, I}
{J, G}
{A, B}
{A, D}
{E, F}

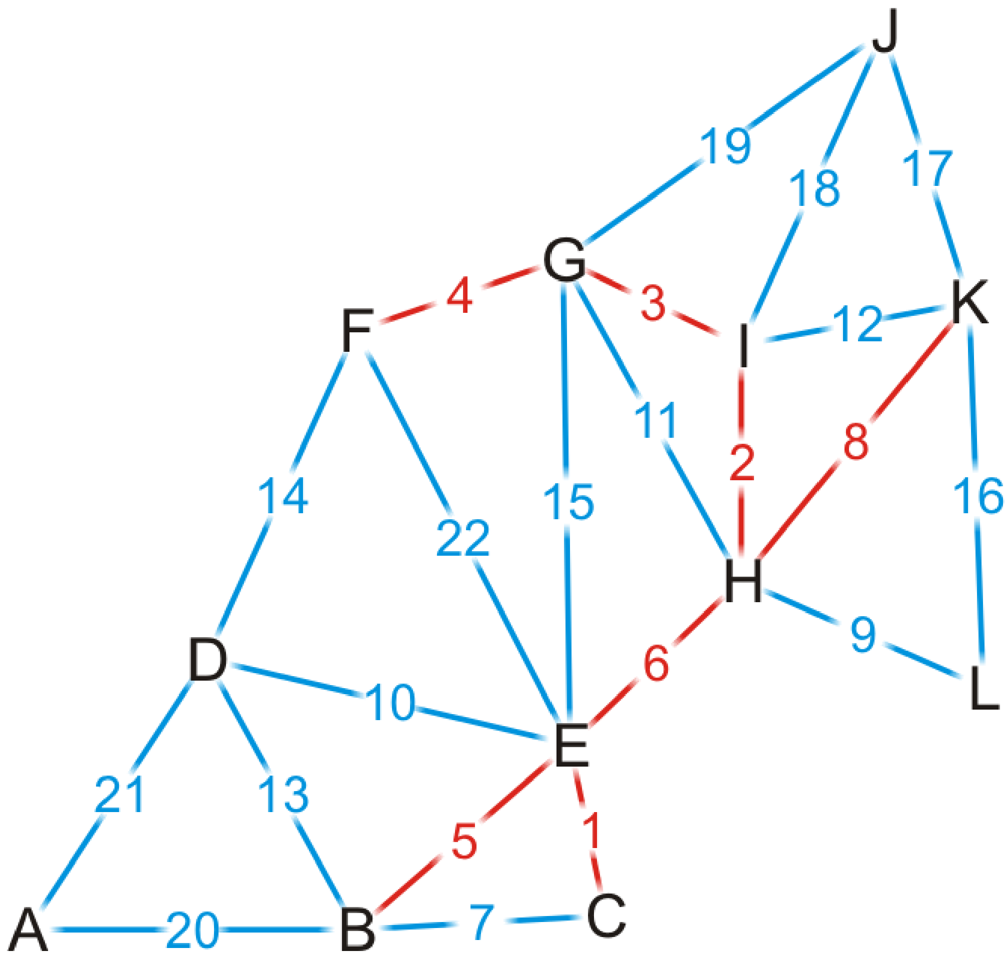
Kruskal's Algorithm – Example

- We try adding {B, C}, but it creates a cycle



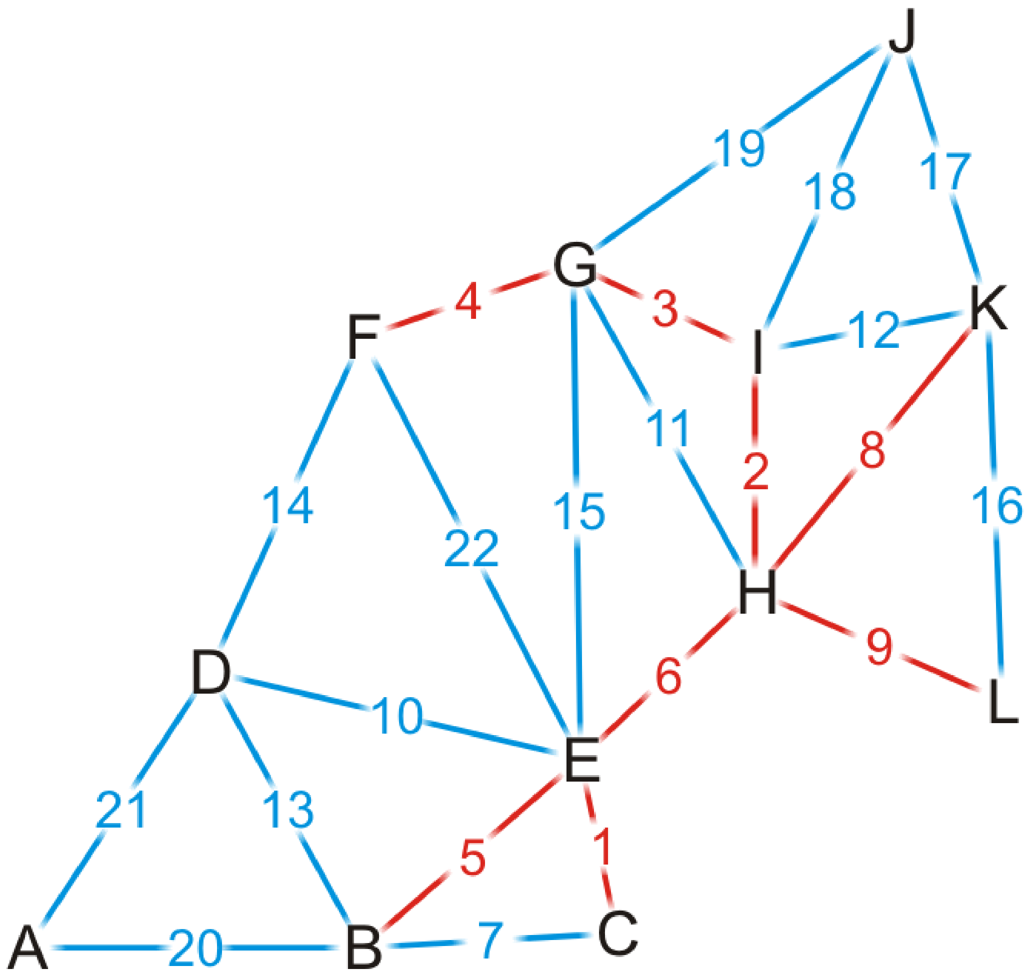
{C, E}
{H, I}
{G, I}
{F, G}
{B, E}
{E, H}
{B, C}
{H, K}
{H, L}
{D, E}
{G, H}
{I, K}
{B, D}
{D, F}
{E, G}
{K, L}
{J, K}
{J, I}
{J, G}
{A, B}
{A, D}
{E, F}

Kruskal's Algorithm – Example



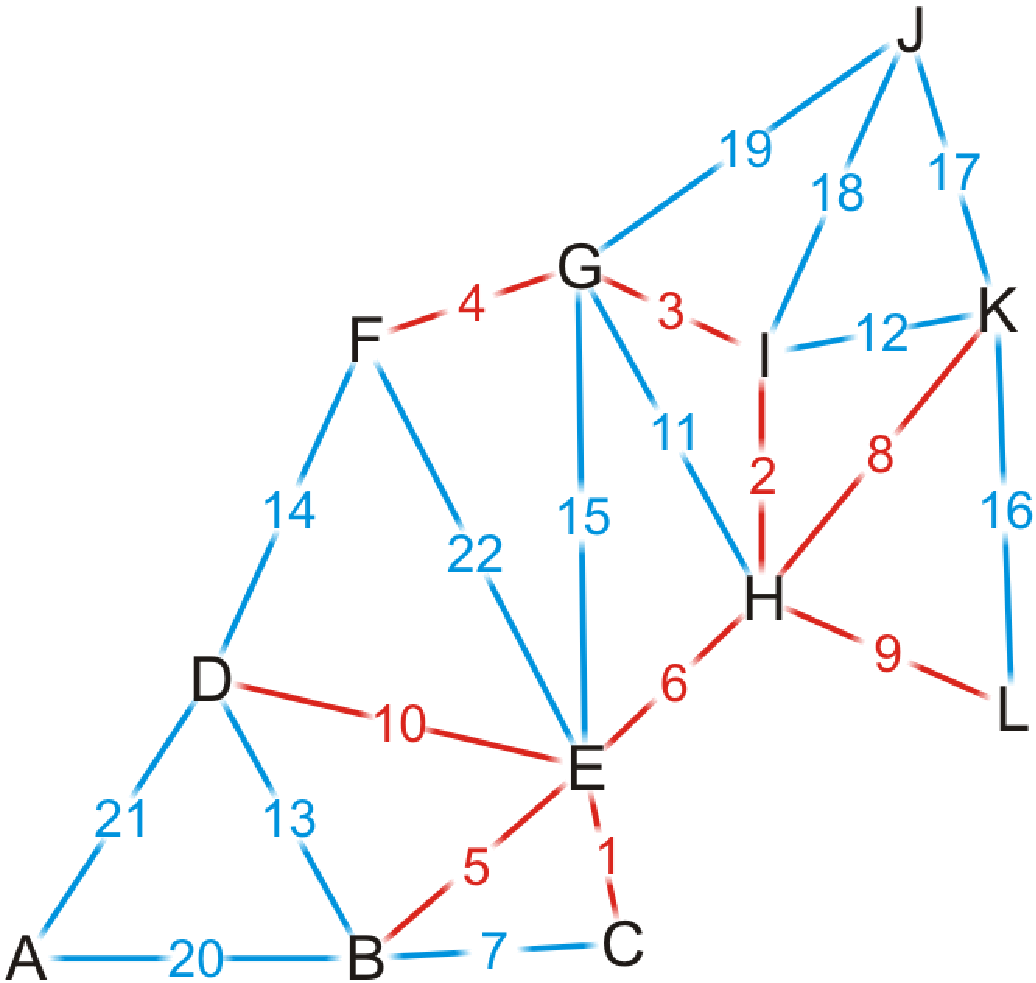
{C, E}
{H, I}
{G, I}
{F, G}
{B, E}
{E, H}
{B, C}
{H, K}
{H, L}
{D, E}
{G, H}
{I, K}
{B, D}
{D, F}
{E, G}
{K, L}
{J, K}
{J, I}
{J, G}
{A, B}
{A, D}
{E, F}

Kruskal's Algorithm – Example



{C, E}
{H, I}
{G, I}
{F, G}
{B, E}
{E, H}
{B, C}
{H, K}
{H, L}
{D, E}
{G, H}
{I, K}
{B, D}
{D, F}
{E, G}
{K, L}
{J, K}
{J, I}
{J, G}
{A, B}
{A, D}
{E, F}

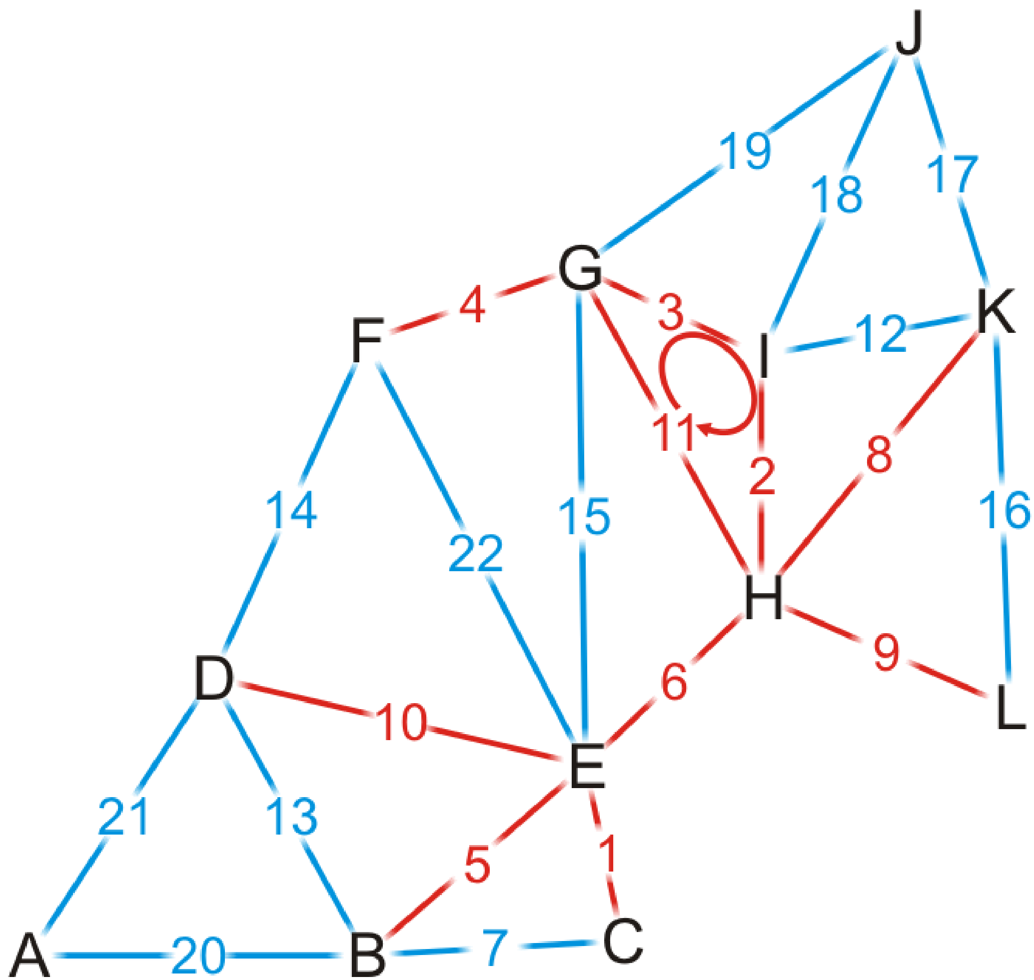
Kruskal's Algorithm – Example



{C, E}
{H, I}
{G, I}
{F, G}
{B, E}
{E, H}
{B, C}
{H, K}
{H, L}
{D, E}
{G, H}
{I, K}
{B, D}
{D, F}
{E, G}
{K, L}
{J, K}
{J, I}
{J, G}
{A, B}
{A, D}
{E, F}

Kruskal's Algorithm – Example

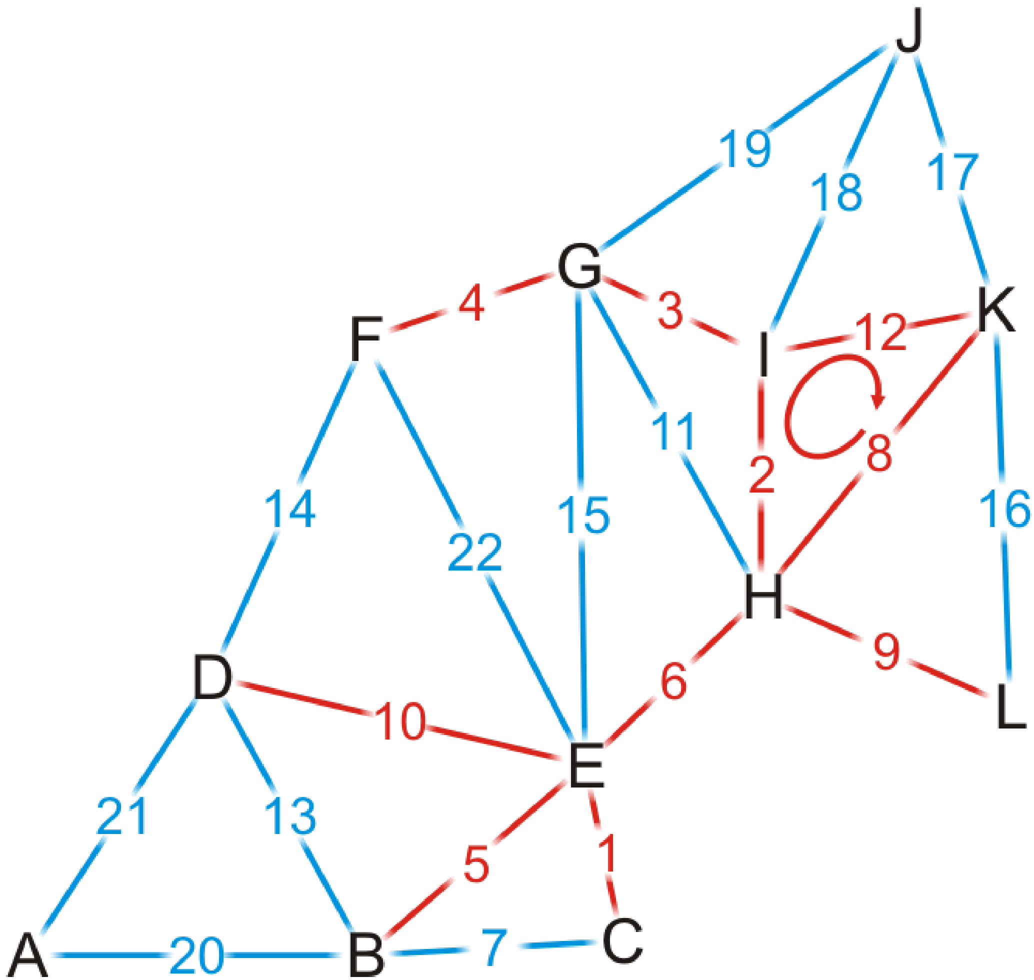
- We try adding {G, H}, but it creates a cycle



{C, E}
{H, I}
{G, I}
{F, G}
{B, E}
{E, H}
{B, C}
{H, K}
{H, L}
{D, E}
{G, H}
{I, K}
{B, D}
{D, F}
{E, G}
{K, L}
{J, K}
{J, I}
{J, G}
{A, B}
{A, D}
{E, F}

Kruskal's Algorithm – Example

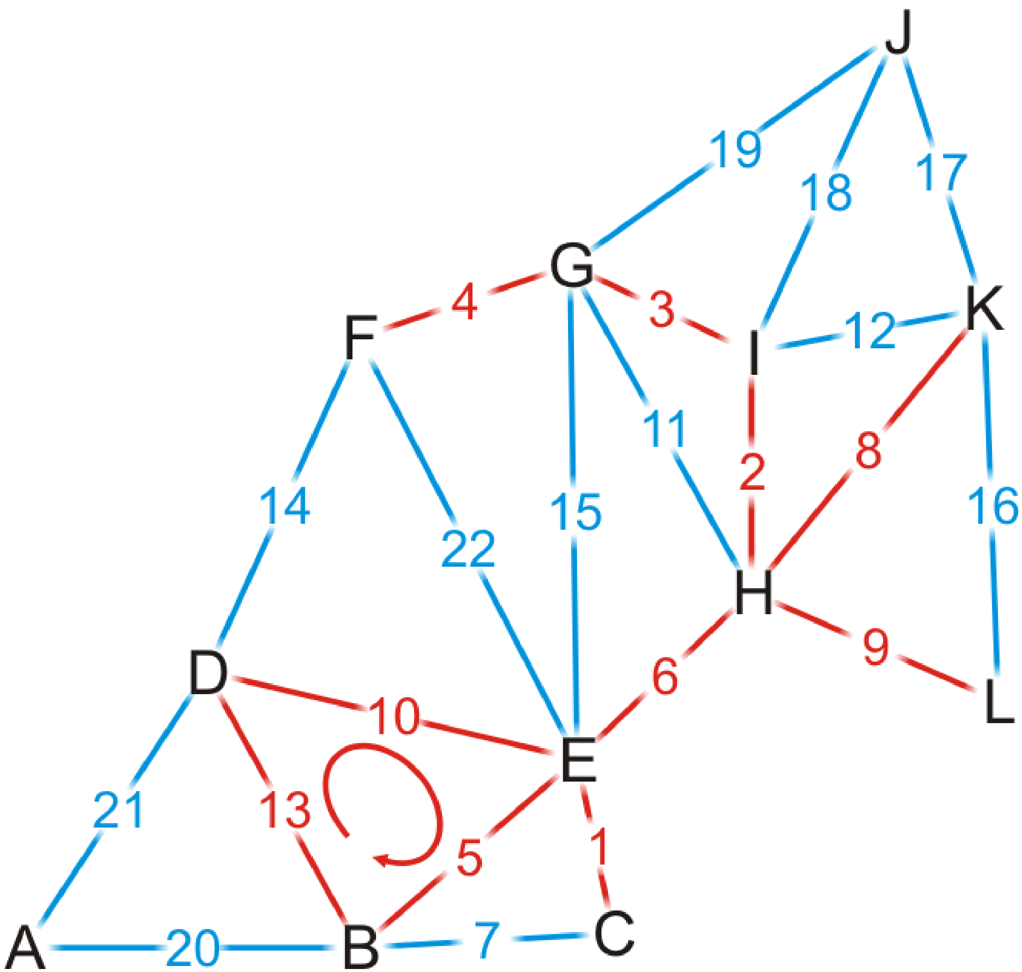
- We try adding {I, K}, but it creates a cycle



{C, E}
{H, I}
{G, I}
{F, G}
{B, E}
{E, H}
{B, C}
{H, K}
{H, L}
{D, E}
{G, H}
{I, K}
{B, D}
{D, F}
{E, G}
{K, L}
{J, K}
{J, I}
{J, G}
{A, B}
{A, D}
{E, F}

Kruskal's Algorithm – Example

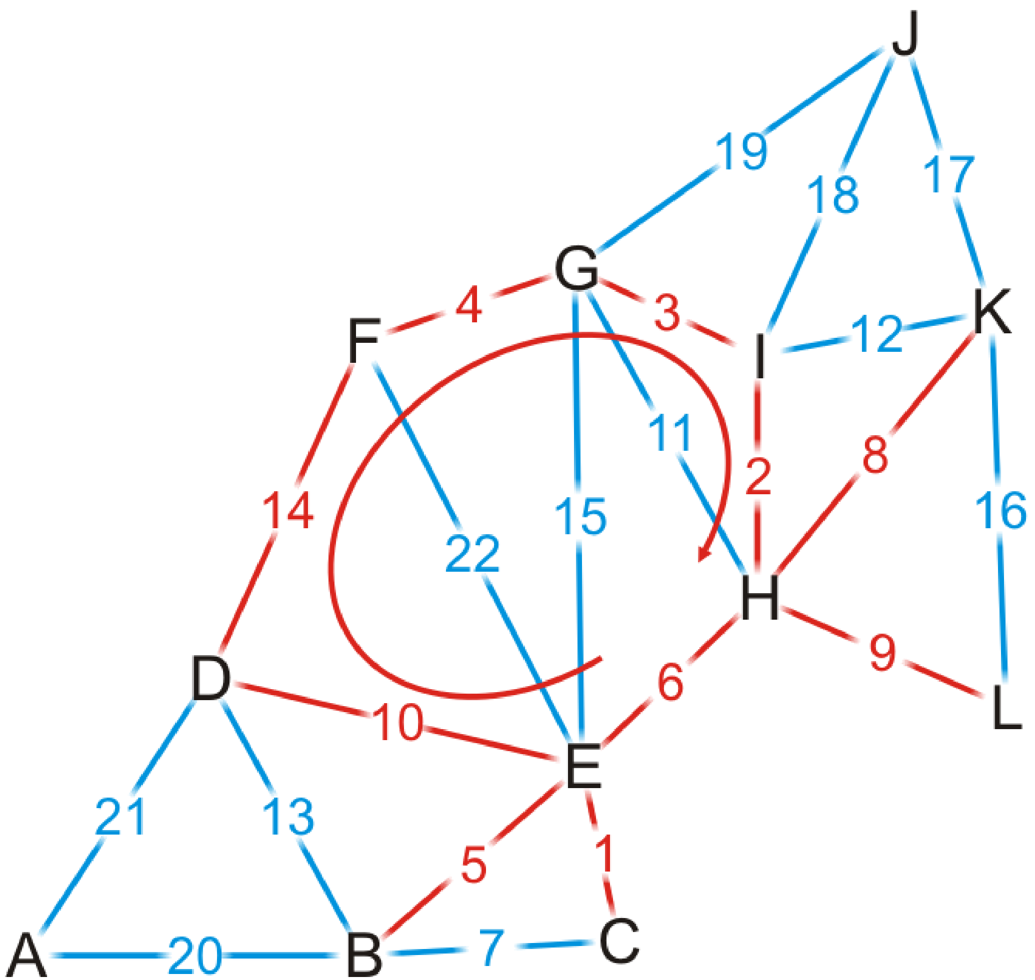
- We try adding {B, D}, but it creates a cycle



{C, E}
{H, I}
{G, I}
{F, G}
{B, E}
{E, H}
{B, C}
{H, K}
{H, L}
{D, E}
{G, H}
{I, K}
{B, D}
{D, F}
{E, G}
{K, L}
{J, K}
{J, I}
{J, G}
{A, B}
{A, D}
{E, F}

Kruskal's Algorithm – Example

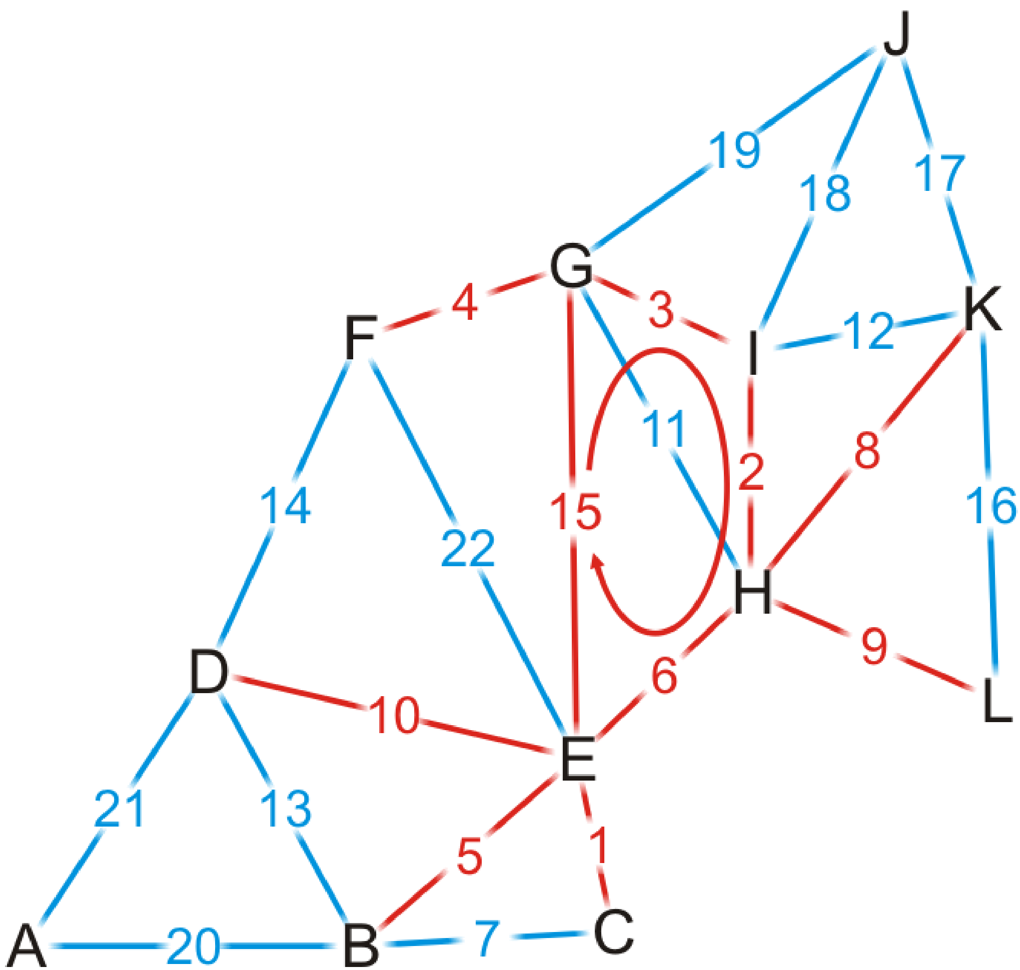
- We try adding {D, F}, but it creates a cycle



{C, E}
{H, I}
{G, I}
{F, G}
{B, E}
{E, H}
{B, C}
{H, K}
{H, L}
{D, E}
{G, H}
{I, K}
{B, D}
{D, F}
{E, G}
{K, L}
{J, K}
{J, I}
{J, G}
{A, B}
{A, D}
{E, F}

KRUSKAL'S ALGORITHM – EXAMPLE

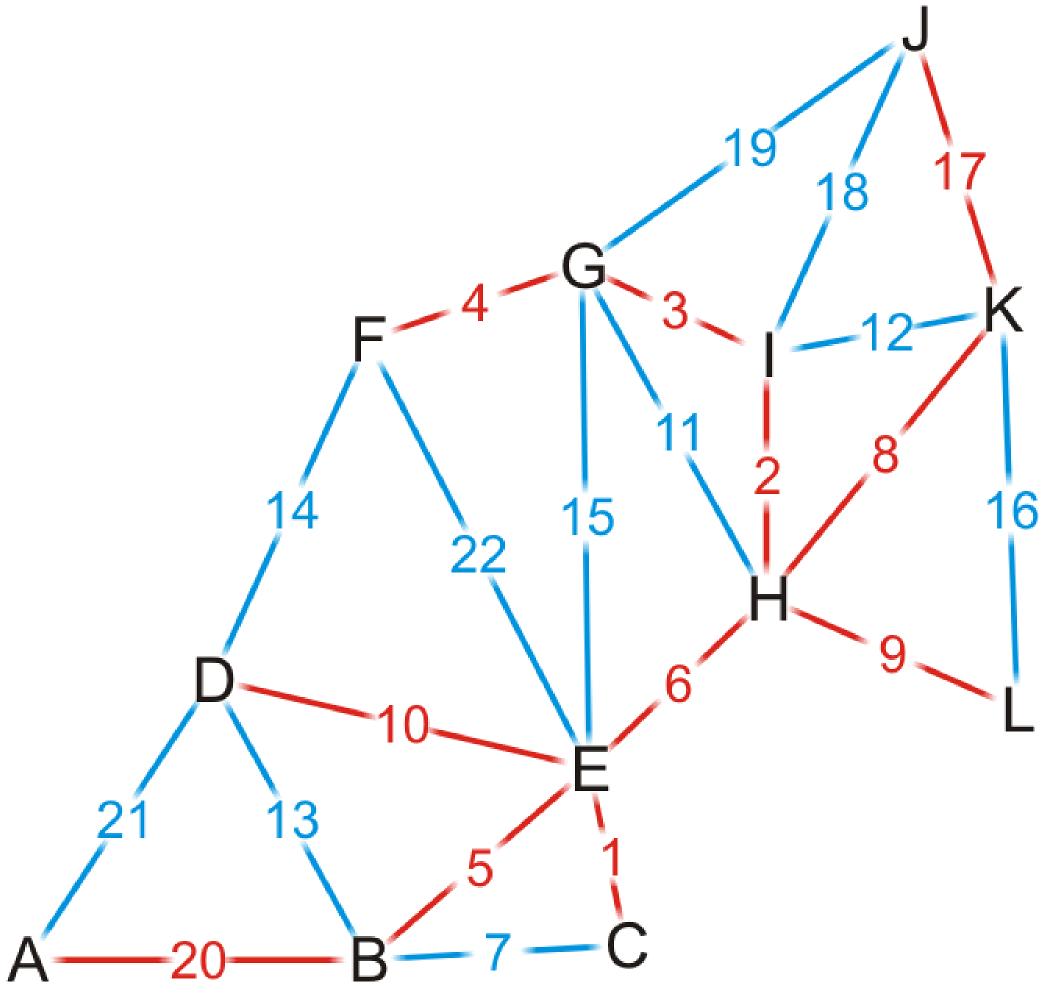
- We try adding {E, G}, but it creates a cycle



{C, E}
{H, I}
{G, I}
{F, G}
{B, E}
{E, H}
{B, C}
{H, K}
{H, L}
{D, E}
{G, H}
{I, K}
{B, D}
{D, F}
{E, G}
{K, L}
{J, K}
{J, I}
{J, G}
{A, B}
{A, D}
{E, F}

Kruskal's Algorithm – Example

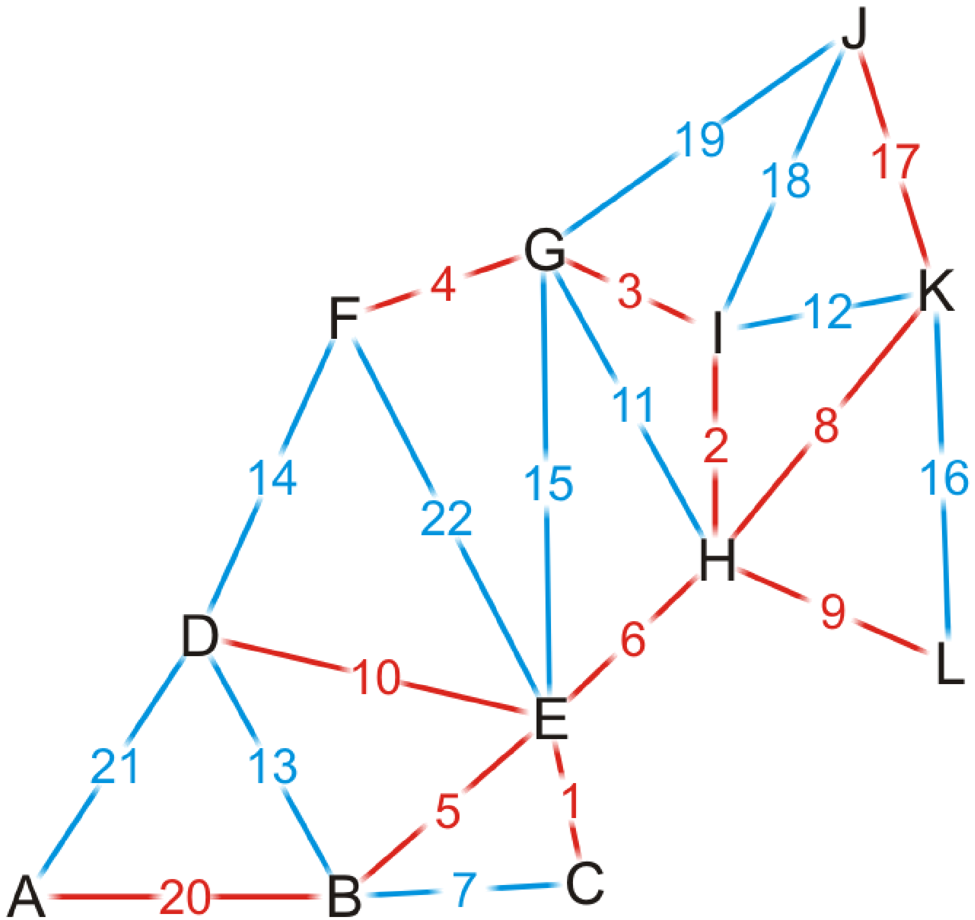
- By observation, we can still add edges {J, K} and {A, B}



{C, E}
{H, I}
{G, I}
{F, G}
{B, E}
{E, H}
{B, C}
{H, K}
{H, L}
{D, E}
{G, H}
{I, K}
{B, D}
{D, F}
{E, G}
{K, L}
{J, K}
{J, I}
{J, G}
{A, B}
{A, D}
{E, F}

Kruskal's Algorithm – Example

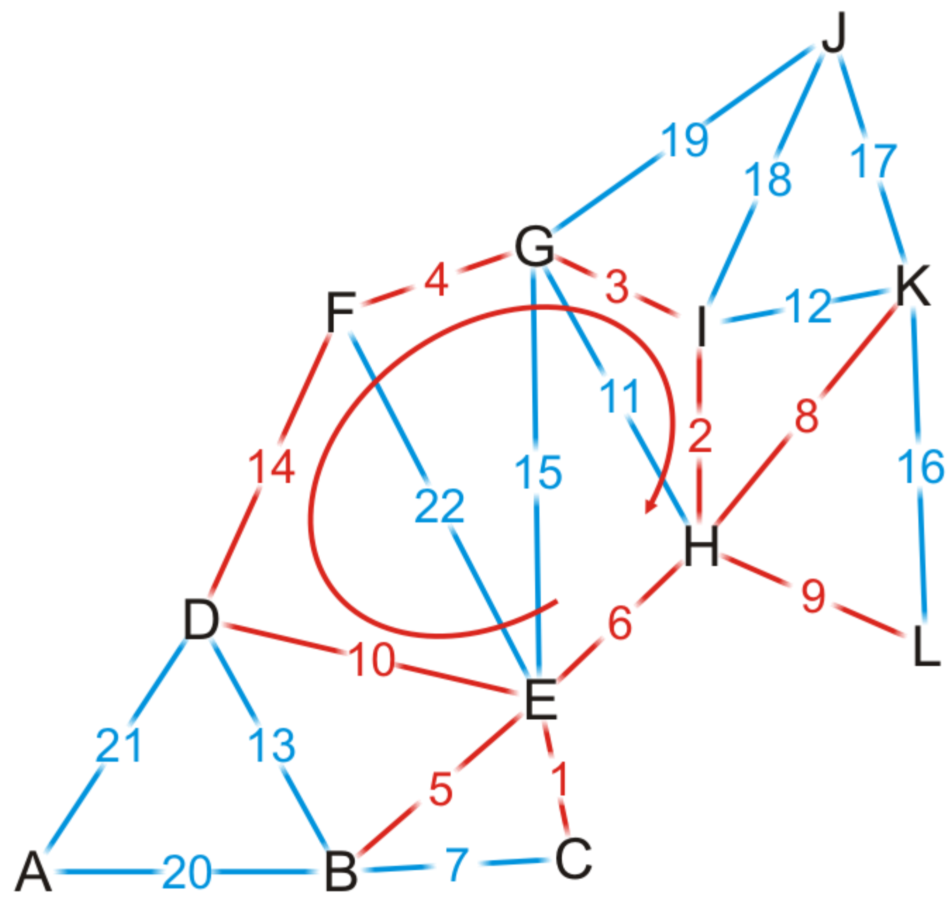
- Having added {A, B}, we now have 11 edges
  - We terminate the loop
  - We have our minimum spanning tree



{C, E}
{H, I}
{G, I}
{F, G}
{B, E}
{E, H}
{B, C}
{H, K}
{H, L}
{D, E}
{G, H}
{I, K}
{B, D}
{D, F}
{E, G}
{K, L}
{J, K}
{J, I}
{J, G}
{A, B}
{A, D}
{E, F}

Detecting a Cycle

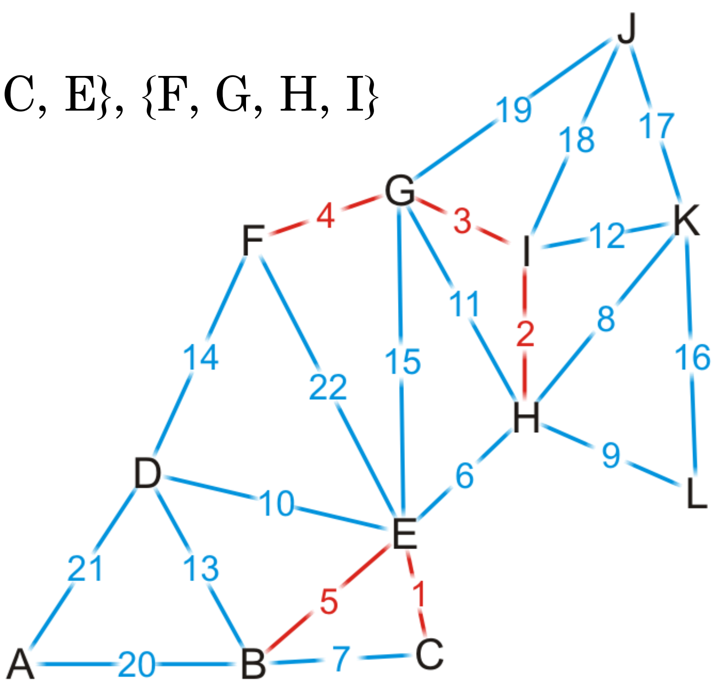
- To determine if a cycle is created, we could perform a traversal
  - A run-time of `O(|V|)`



Detecting a Cycle – Disjoint Sets

- Consider edges in the same connected sub-graph as forming a set

{B, C, E}, {F, G, H, I}

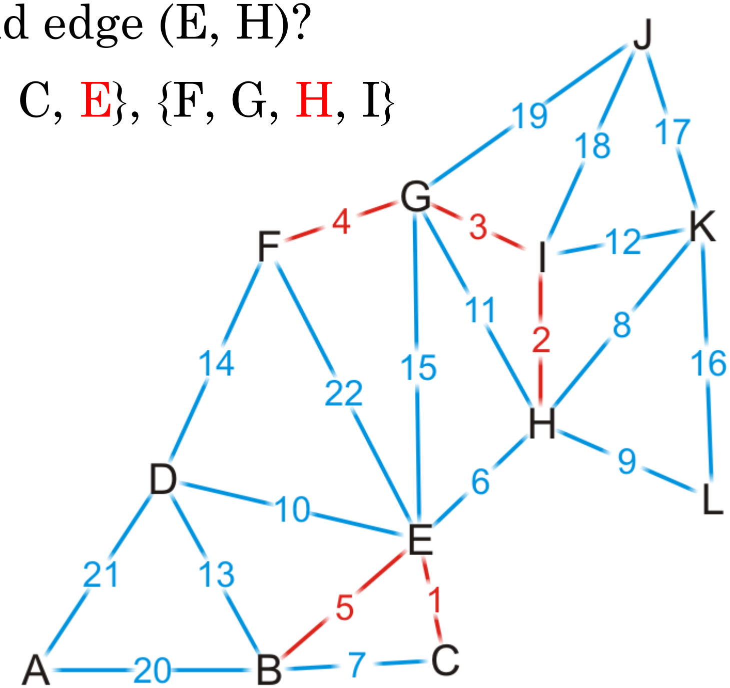


Detecting a Cycle – Disjoint Sets

- Consider edges in the same connected sub-graph as forming a set
- If the vertices of the next edge are in different sets
  - Take the union of the two sets

Add edge (E, H)?

{B, C, E}, {F, G, H, I}

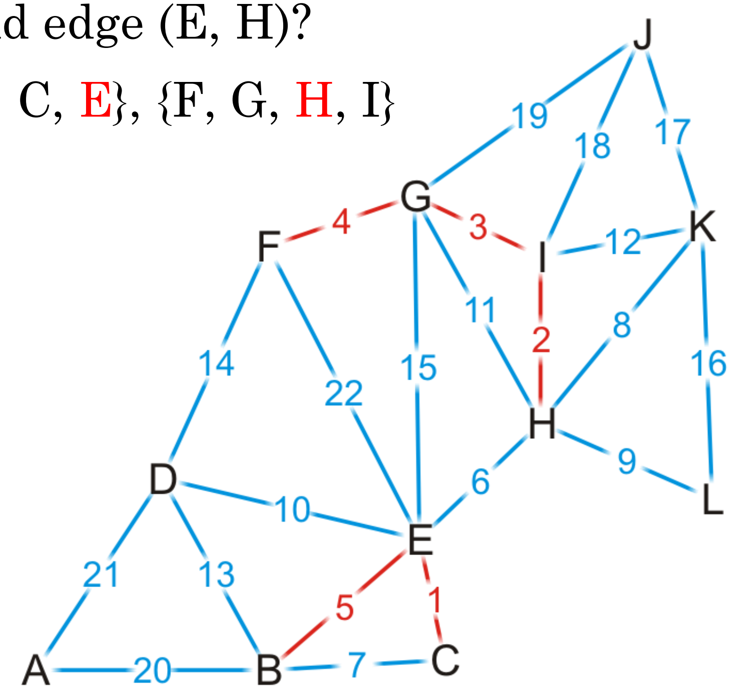


Detecting a Cycle – Disjoint Sets

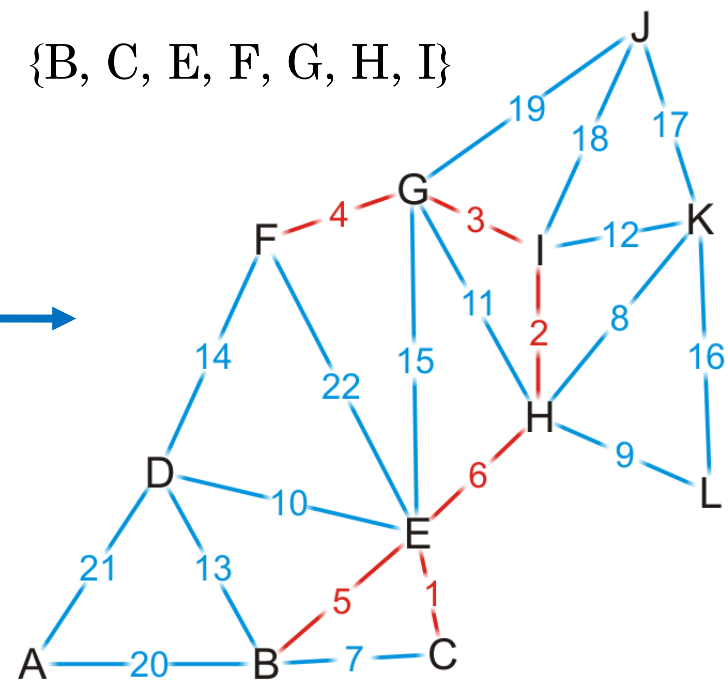
- Consider edges in the same connected sub-graph as forming a set
- If the vertices of the next edge are in different sets
  - Take the union of the two sets

Add edge (E, H)?

{B, C, E}, {F, G, H, I}



{B, C, E, F, G, H, I}

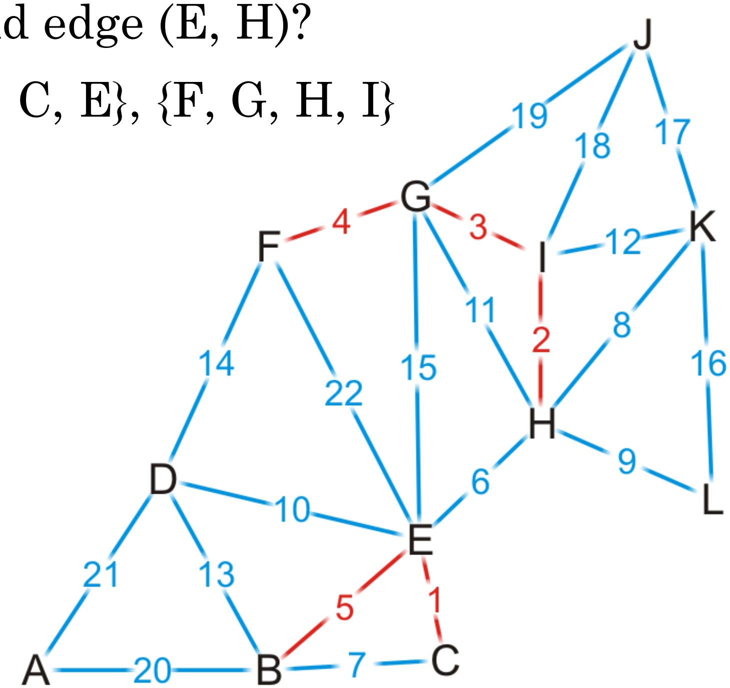


Detecting a Cycle – Disjoint Sets

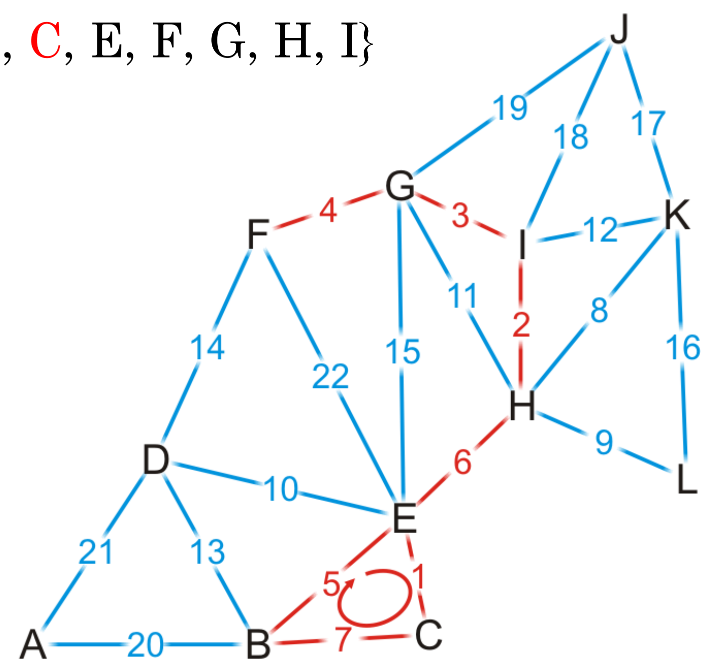
- Consider edges in the same connected sub-graph as forming a set
- If the vertices of the next edge are in different sets
  - Take the union of the two sets
- Do not add an edge if both vertices are in the same set

Add edge (E, H)?

{B, C, E}, {F, G, H, I}



{B, C, E, F, G, H, I}



KRUSKAL(V, E, w)

```
A ← ∅
for each vertex v ∈ V                                   O(V)
        do MAKE-SET(v)
sort E into non-decreasing order by w                   O(ElgE)
for each (u, v) taken from the sorted list              O(E)
        do if FIND-SET(u) ≠ FIND-SET(v)
                then A ← A ∪ {(u, v)}
                        UNION(u, v)                     O(lgV)
return A
```

Running time: O(V+ElgE+ElgV)=O(ElgE) – dependent on the implementation of the disjoint-set data structure

KRUSKAL(V, E, w) (CONT.)

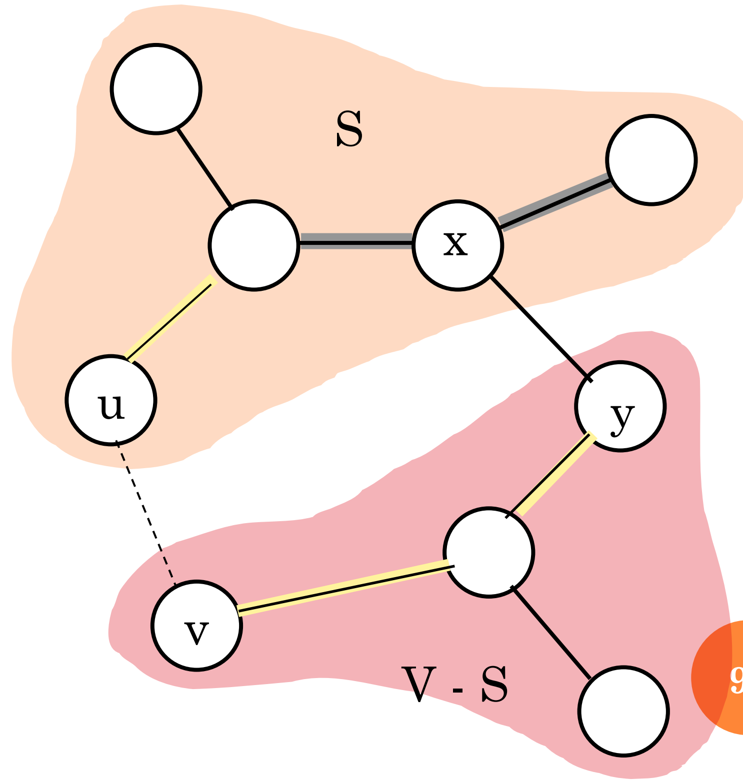
```
A ← ∅
for each vertex v ∈ V                                  O(V)
        do MAKE-SET(v)
sort E into non-decreasing order by w                  O(ElgE)
for each (u, v) taken from the sorted list             O(E)
        do if FIND-SET(u) ≠ FIND-SET(v)
                then A ← A ∪ {(u, v)}
                        UNION(u, v)                    O(lgV)
return A
```

- Running time: $O(V+ElgE+ElgV)=O(ElgE)$
- Since $E=O(V^2)$, we have $lgE=O(2lgV)=O(lgV)$

O(ElgV)

Kruskal's Algorithm

- Kruskal's algorithm is a **"greedy"** algorithm
- Kruskal's greedy strategy produces a globally optimum solution
- Proof for generic approach applies to Kruskal's algorithm too



Kruskal's Algorithm vs Prim's Algorithm

- In **prim's algorithm**, graph must be a **connected**
- **Kruskal's** algorithm can function on **disconnected** graphs too
- **Prim's algorithm** is significantly faster for **dense graphs** with more number of edges than vertices
- **Kruskal's algorithm** runs faster in the case of **sparse graphs**

Problem 1

- **(Exercise 23.2-3, page 637**) Compare Prim's algorithm with and Kruskal's algorithm assuming:

(a) sparse graphs:

In this case, E=O(V)

Kruskal:

O(ElgE)=O(VlgV)

Prim:

- binary heap: O(ElgV)=O(VlgV)
- Fibonacci heap: O(VlgV+E)=O(VlgV)

Problem 1 (cont.)

(b) dense graphs

In this case, $E=O(V^2)$

Kruskal:

$O(ElgE)=O(V^2lgV^2)=O(2V^2lgV)=O(V^2lgV)$

Prim:

- binary heap: $O(ElgV)=O(V^2lgV)$
- Fibonacci heap: $O(VlgV+E)=O(VlgV+V^2)=O(V^2)$

Problem 2

- Suppose that some of the weights in a connected graph G are negative. Will Prim's algorithm still work? What about Kruskal's algorithm? Justify your answers.
  - Yes, both algorithms will work with negative weights. Review the proof of the generic approach; there is no assumption in the proof about the weights being positive.

Problem 3

- Find an algorithm for the "maximum" spanning tree. That is, given an undirected weighted graph G, find a spanning tree of G of maximum cost. Prove the correctness of your algorithm.
  - Consider choosing the "heaviest" edge (i.e., the edge associated with the largest weight) in a cut. The generic proof can be modified easily to show that this approach will work.
  - Alternatively, multiply the weights by -1 and apply either Prim's or Kruskal's algorithms without any modification at all!

MST ALGORITHMS

Kruskal's algorithm (see CLRS):

- Uses the ***disjoint-set data structure*** (see CLRS, Ch. 21).
- Running time $= O(E \lg V)$.

Best to date:

- Karger, Klein, and Tarjan [1993].
- Randomized algorithm.
- $O(V + E)$ expected time.

Reference

- **Introduction to Algorithms**
  - Minimum Spanning Tree
  - Chapter # 23
  - Thomas H. Cormen

  - Disjoint Set
  - http://www.csl.mtu.edu/cs4321/www/Lectures/Lecture%2019%20-%20Kruskal%20Algorithm%20and%20Dis-joint%20Sets.htm

Summary of Graph Algortihms

Breadth-first search

BFS(G, s)
for each vertex $u \; V[G] - \{s\}$
  do $color[u] \leftarrow$ WHITE
    $d[u] \leftarrow \infty$
    $\pi[u] \leftarrow$ NIL
$color[s] \leftarrow$ GRAY
$d[s] \leftarrow 0$
$\pi[s] \leftarrow$ NIL
$Q \leftarrow \varnothing$
ENQUEUE(Q, s)
while $Q \neq \varnothing$
  do $u \leftarrow$ DEQUEUE(Q)
    for each $v \in Adj[u]$
      do if $color[v]$ = WHITE
        then $color[v] \leftarrow$ GRAY
          $d[v] \leftarrow d[u] + 1$
          $\pi[v] \leftarrow u$
          ENQUEUE(Q, v)
$color[u] \leftarrow$ BLACK

Depth-first search

DFS(G)
for each vertex $u \in G.V$
  $u.color$ = WHITE
  $u.\pi$ = NIL
$time = 0$
for each vertex $u \in G.V$
  if $u.color$ == WHITE
    DFS-VISIT(G, u)

DFS-VISIT(G, u)
$time = time + 1$ // whi
$u.d = time$
$u.color$ = GRAY
for each $v \in G.Adj[u]$ // exp
  if $v.color$ == WHITE
    $v.\pi = u$
    DFS-VISIT(G, v)
$u.color$ = BLACK // blac
$time = time + 1$
$u.f = time$

Analysis of Prim

$$\left.\begin{array}{c}\Theta(V)\\ \text{total}\end{array}\right\{\begin{array}{l} Q \leftarrow V \\ key[v] \leftarrow \infty \text{ for all } v \in V \\ key[s] \leftarrow 0 \text{ for some arbitrary } s \in V \end{array}$$

$$\left.\begin{array}{c}|V|\\ \text{times}\end{array}\right\{\begin{array}{l} \textbf{while } Q \neq \emptyset \\ \quad \textbf{do } u \leftarrow \text{EXTRACT-MIN}(Q) \\ \quad \left.\begin{array}{c} degree(u) \\ \text{times}\end{array}\right\{\begin{array}{l} \textbf{for} \text{ each } v \in Adj[u] \\ \quad \textbf{do if } v \in Q \text{ and } w(u,v) < key[v] \\ \qquad \textbf{then } key[v] \leftarrow w(u,v) \\ \qquad\qquad \pi[v] \leftarrow u \end{array}\end{array}$$

Handshaking Lemma $\Rightarrow \Theta(E)$ implicit DECREASE-KEY's.

Time $= \Theta(V) \cdot T_{\text{EXTRACT-MIN}} + \Theta(E) \cdot T_{\text{DECREASE-KEY}}$

KRUSKAL(V, E, w) (CONT.)

```
A ← ∅                                                  } O(V)
for each vertex v ∈ V
        do MAKE-SET(v)                                 } O(ElgE)
sort E into non-decreasing order by w
for each (u, v) taken from the sorted list              ← O(E)
      do if FIND-SET(u) ≠ FIND-SET(v)
              then A ← A ∪ {(u, v)}                     } ← O(lgV)
                      UNION(u, v)
return A
```

- Running time: O(V+ElgE+ElgV)=O(ElgE)
- Since $E=O(V^2)$, we have lgE=O(2lgV)=O(lgV) } O(ElgV)

Shortest Path