[SWE2015-41] Introduction to Data Structures (자료구조개론)

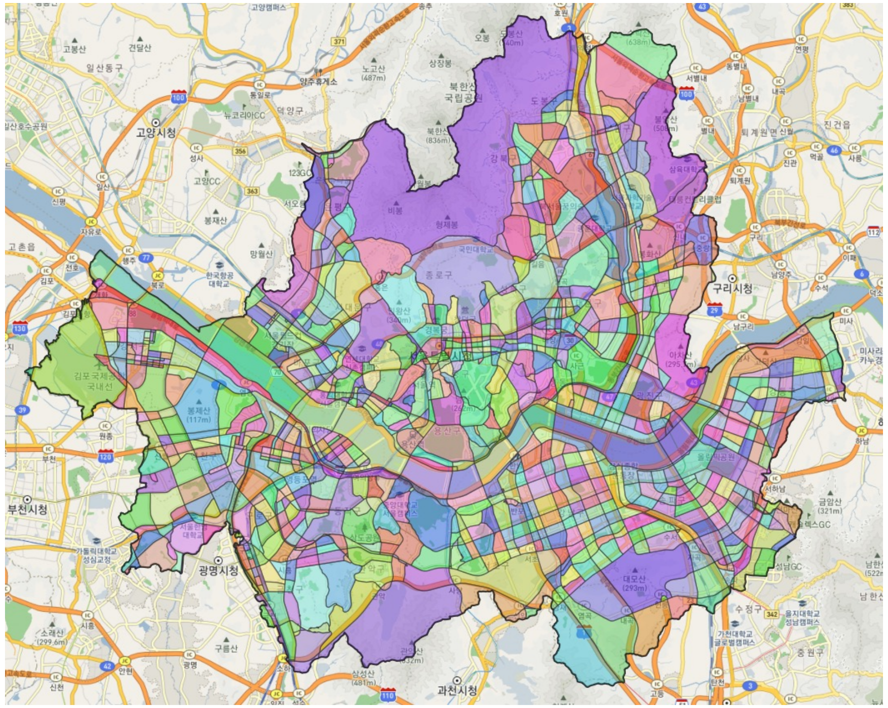
Minimum Spanning Trees

Department of Computer Science and Engineering

Instructor: Hankook Lee (이한국)

(Recap) Graphs

- A graph is a collection of **vertices** and **edges** that connect these vertices
  - Example: social networks, maps



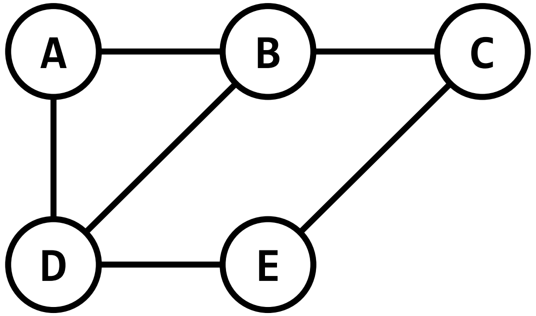
Social Networks
- Each vertex represents a person
- Each edge represents a relationship between two people

Maps
- Each vertex represents a building
- Each edge represents a street between two buildings

(Recap) Undirected Graphs

- A graph is a collection of **vertices** and **edges** that connect these vertices
  - Example: social networks, maps

- **Definition:** A graph G is defined by V and E, i.e., G=(V,E) where
  - V is a set of vertices in G
  - E is a set of edges in G
  - An edge $(u,v) \in E$ is a pair of two vertices $u,v \in V$
  - **Undirected graph**: vertices can be traversed from u to v as well as from v to u
    - The order between u and v is not important



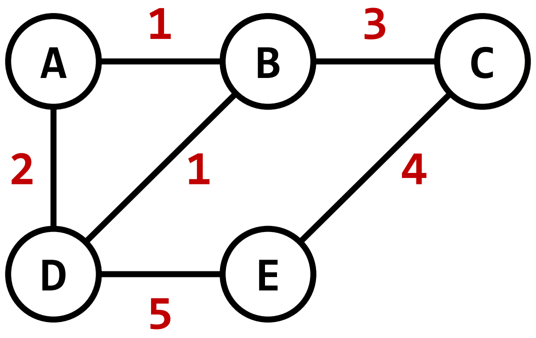
V = { A, B, C, D, E }

E = { (A,B), (A,D), (B,C),
(B,D), (C,E), (D,E), }

(Recap) Weighted Graphs

- A graph is a collection of **vertices** and **edges** that connect these vertices
  - Example: social networks, maps

- **Definition:** A graph `G` is defined by `V` and `E`, i.e., `G=(V,E)` where
  - `V` is a set of vertices in `G`
  - `E` is a set of edges in `G`
  - An edge `(u,v)` ∈ `E` is a pair of two vertices `u,v` ∈ `V`
  - **Weighted graphs**: each edge is associated with a **weight** value
    - The edge representation `(u,v)` is extended to `(u,v,w)` where `w` is the weight of `(u,v)`




```
V = { A, B, C, D, E }

E = { (A,B,1), (A,D,2), (B,C,3),
      (B,D,1), (C,E,4), (D,E,5), }
```

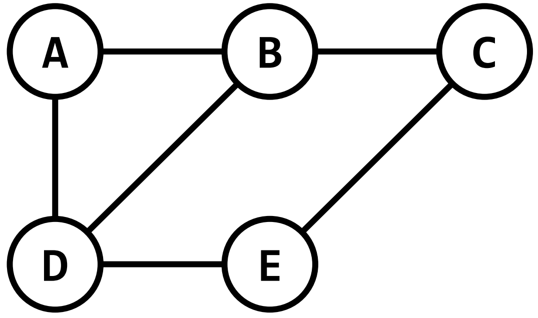
(Recap) Graph Terminology

- **Adjacent** nodes or **Neighbors**
  - For every edge `(u,v)` ∈ E,
  - `u` is said to be **adjacent** to `v` (and vice versa)
  - `u` is said to be a **neighbor** of `v` (and vice versa)

- The **degree** of a vertex `u` (in an undirected graph)
  - the number of edges containing `u` (i.e., `(*,u)` & `(u,*)` edges)

- Examples
  - `A` is **adjacent** to `B` and `D`
  - `B` and `D` are **neighbors** of `A`
  - `degree(A) = 2`
  - `degree(D) = 3`

Undirected Graph

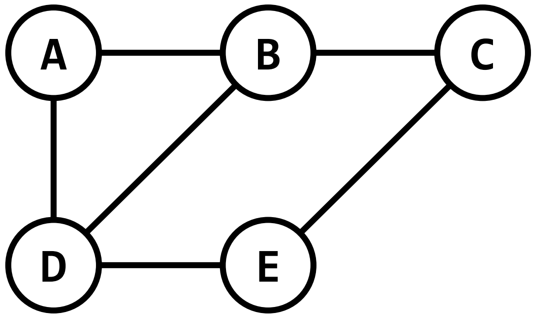


(Recap) Paths

- A **path** P in a graph $G=(V,E)$
  - A sequence of vertices, $P = (v_0, v_1, v_2, ..., v_n)$ where for all i,
    - $(v_i, v_{i+1}) \in E$ or $(v_{i+1}, v_i) \in E$ when G is undirected

For the above path P,

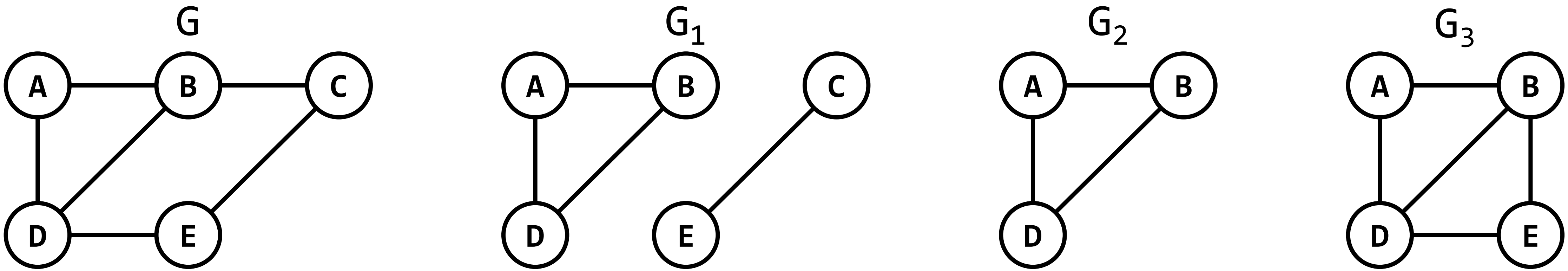
- Its **length** is equal to n, the number of edges on P
- The vertices v_0 and v_n are said to be **connected** by the path P



- (A,B,D,E) is a path
- A and E are connected

(Recap) Subgraphs

- A **subgraph** `G'=(V',E')` of a graph `G=(V,E)` is a graph satisfying …
  - `V'` ⊆ V and `E'` ⊆ E is
  - u ∈ `V'` and v ∈ `V'` for any edge `(u,v)` ∈ `E'`



- G_1 and G_2 subgraphs of G
- G_3 is not a subgraph of G
- G_2 is a subgraph of G_1

Trees (in Graph Theory)

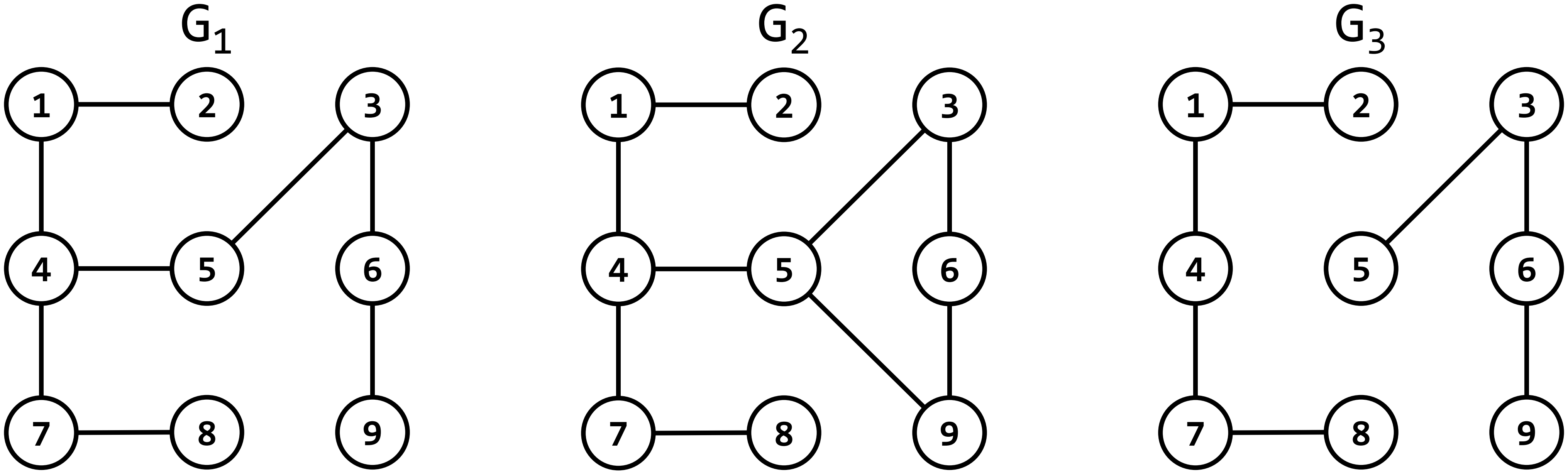
- What is **tree** in **graph theory**?
  - An undirected graph G
  - Any two vertices in G are connected by exactly one simple path

  - **Note.** This definition is slightly different from the tree in computer science …
    - In CS, the tree is directed and rooted
    - In GT, the tree is undirected and has no root

- The following conditions are equivalent to the tree definition
  - G is connected and contains no cycles
  - G is connected, but would become disconnected if any single edge is removed
  - G contains no cycles, and a simple cycle if formed if any edge is added

Trees (in Graph Theory)

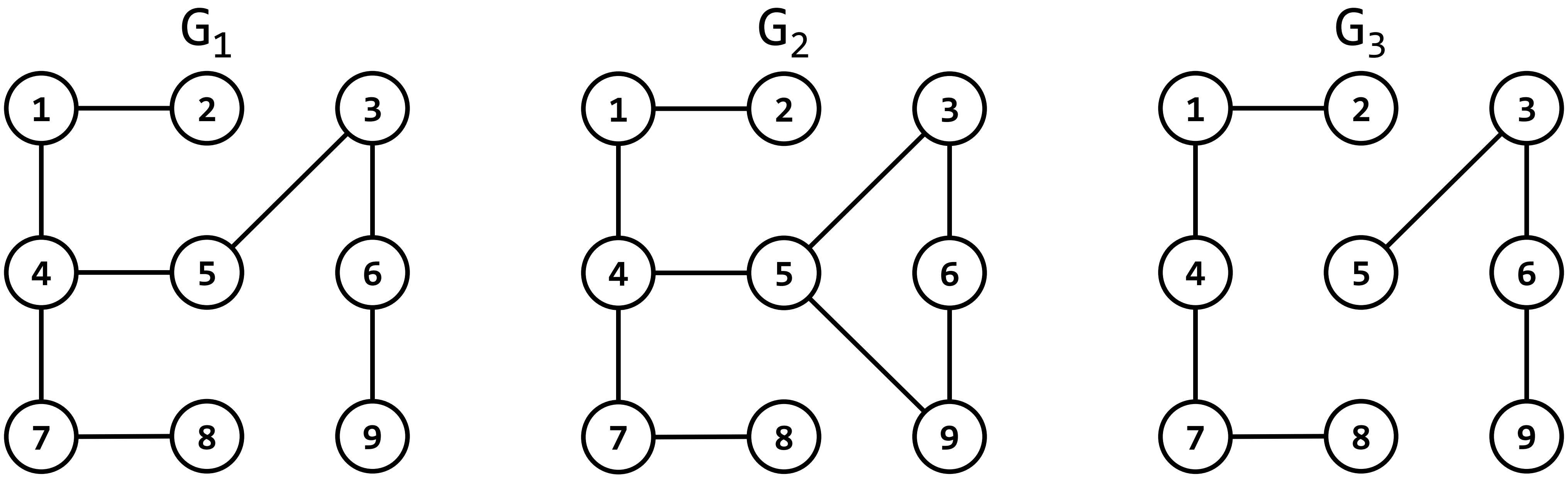
- Which graph is tree?



Trees (in Graph Theory)

- Which graph is tree?

Any two vertices in G are connected by exactly one simple path

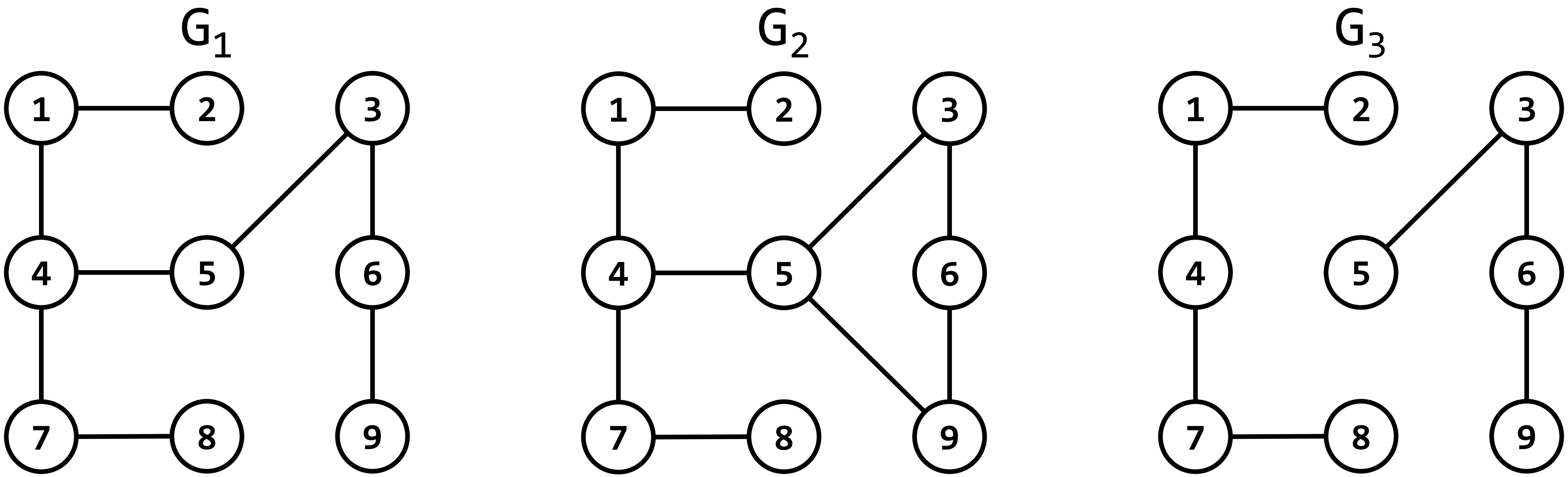


- G_1 is tree because there exists exactly one path between any two vertices
- G_2 is not tree because there exists two paths between **3** and **9**
- G_3 is not tree because there is no path between **1** and **5**

Trees (in Graph Theory)

The graph is connected and contains no cycles

- Which graph is tree?



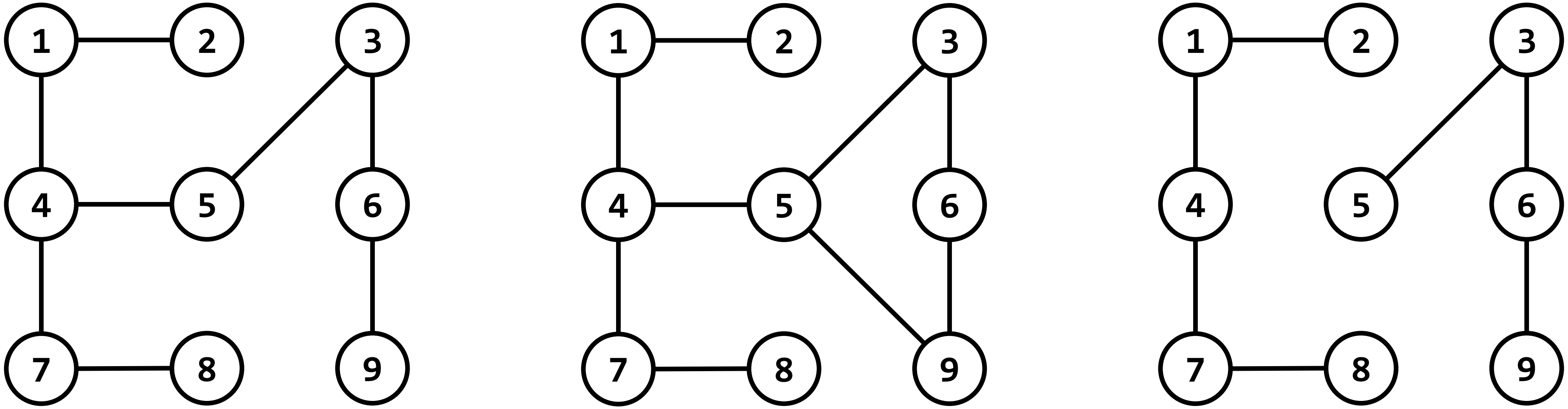
- G_1 is tree because it is connected and contains no cycle
- G_2 is not tree because it contains a cycle $3 \rightarrow 6 \rightarrow 9 \rightarrow 5$
- G_3 is not tree because it is not connected

Trees (in Graph Theory)

- Which graph is tree?

The graph would become disconnected after deleting an edge
A cycle would be formed after adding an edge

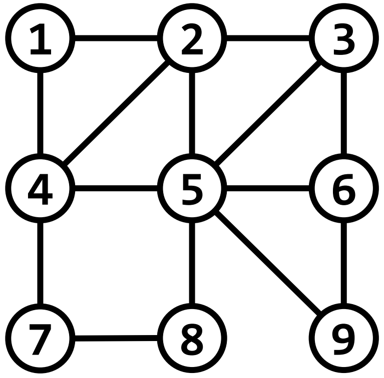
G_1 G_2 G_3



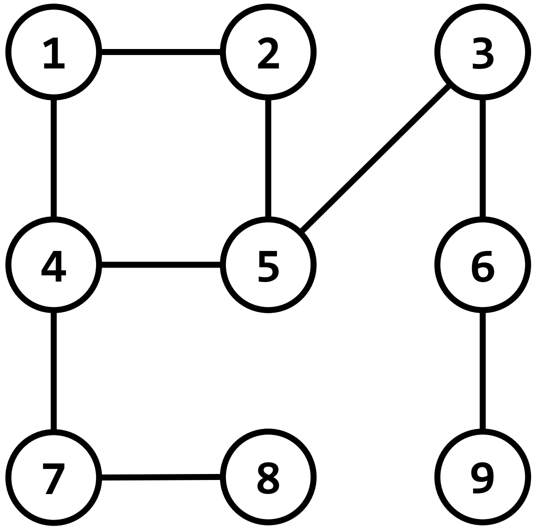
- G_1 would be disconnected after edge deletion & contain a cycle after edge addition
- G_2 is still connected even if the edge $3 \leftrightarrow 5$ is removed from G_2
- G_3 still contains no cycle even if the edge $5 \leftrightarrow 8$ is added to G_3

Spanning Trees

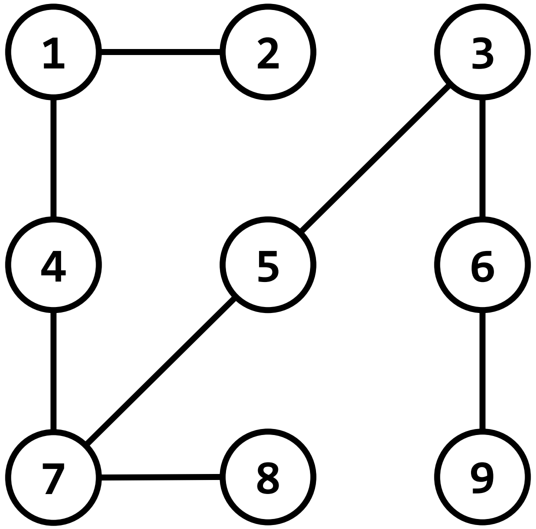
- **A spanning tree T** of an undirected graph **G** is …
  - **T** is a subgraph of **G**
  - **T** is a tree
  - **T** includes all the vertices of **G**



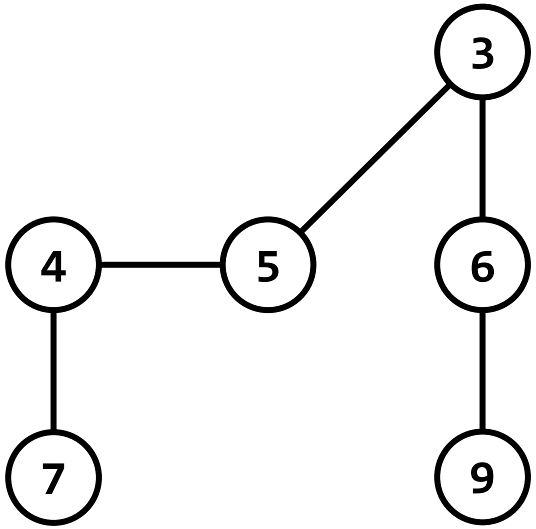
Not a tree



Not a subgraph

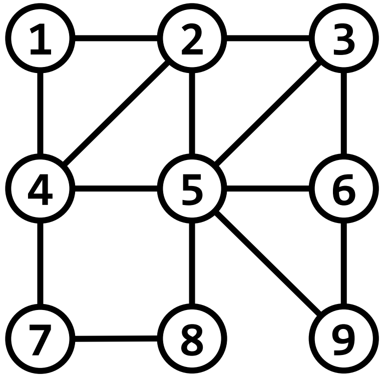


Not include all vertices

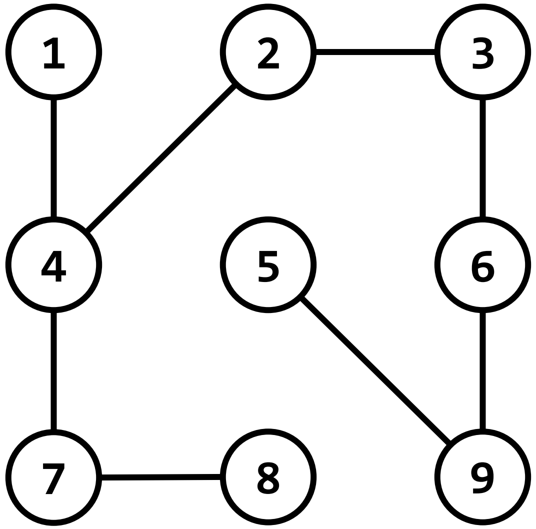


Spanning Trees

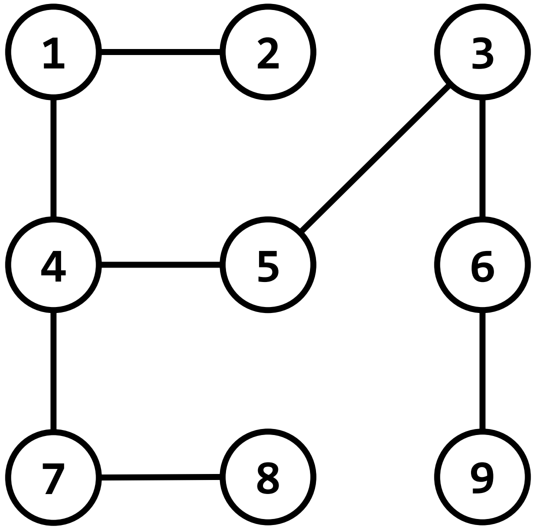
- **A spanning tree T** of an undirected graph **G** is …
  - **T** is a subgraph of **G**
  - **T** is a tree
  - **T** includes all the vertices of **G**



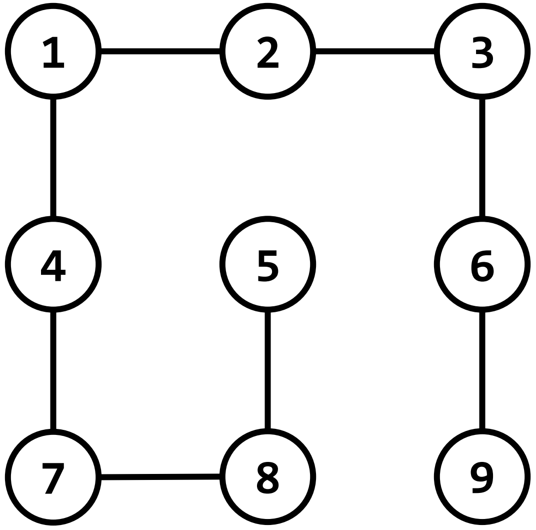
A spanning tree



A spanning tree



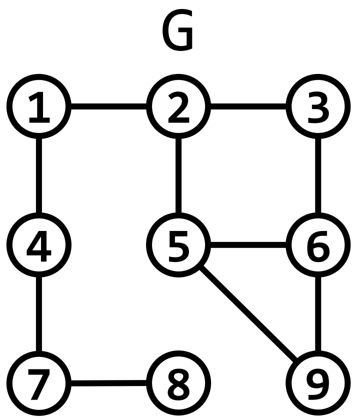
A spanning tree



Spanning Trees

- **A spanning tree T** of an undirected graph **G** is …
  - **T** is a subgraph of **G**
  - **T** is a tree
  - **T** includes all the vertices of **G**

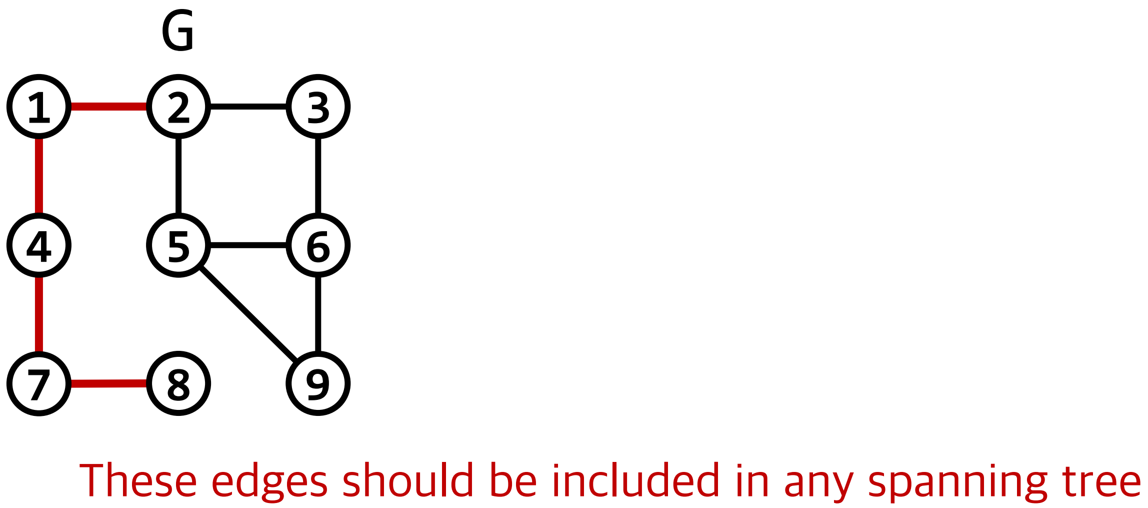
- How many spanning trees do exist in **G**?



Spanning Trees

- **A spanning tree T** of an undirected graph **G** is …
  - **T** is a subgraph of **G**
  - **T** is a tree
  - **T** includes all the vertices of **G**

- How many spanning trees do exist in **G**?

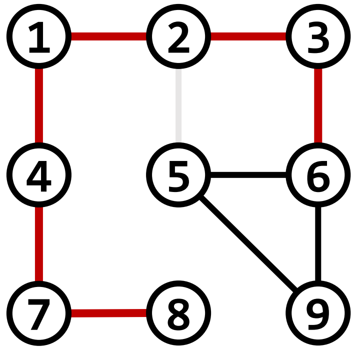


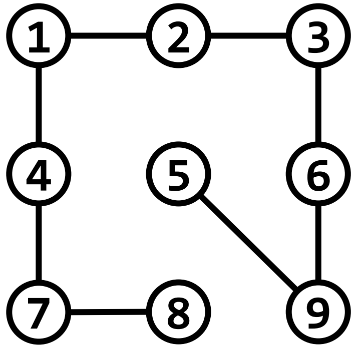
Spanning Trees

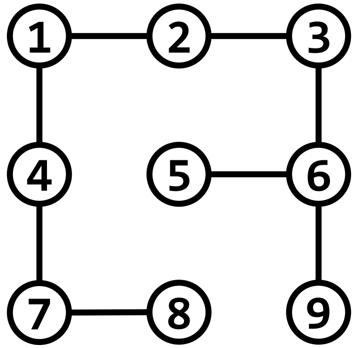
- **A spanning tree T** of an undirected graph **G** is …
  - **T** is a subgraph of **G**
  - **T** is a tree
  - **T** includes all the vertices of **G**

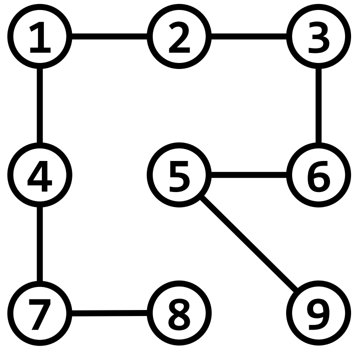
- How many spanning trees do exist in **G**?

G







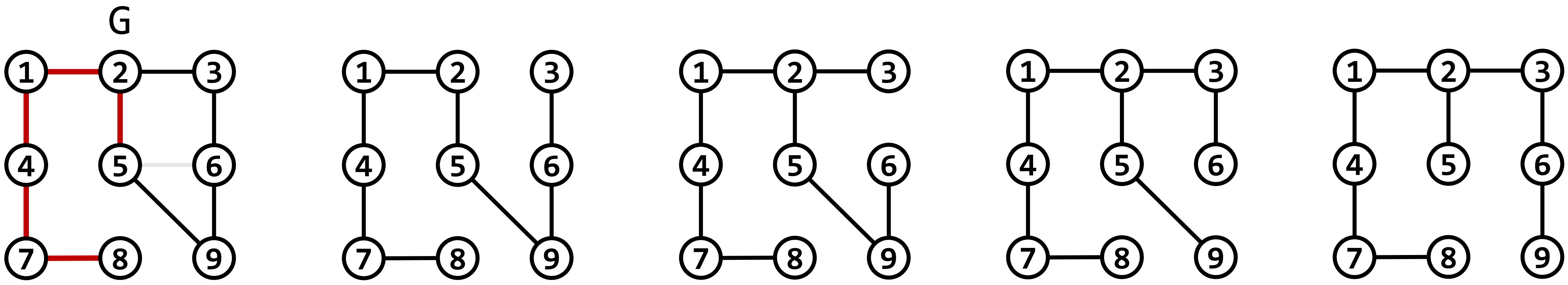


(Case 1) 2 ↔ 5 is not included …

Spanning Trees

- **A spanning tree T** of an undirected graph **G** is …
  - **T** is a subgraph of **G**
  - **T** is a tree
  - **T** includes all the vertices of **G**

- How many spanning trees do exist in **G**?

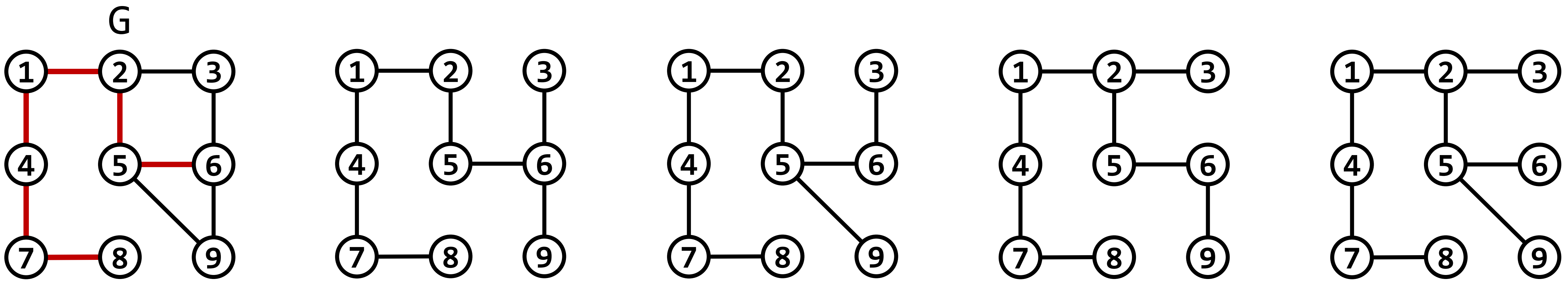


(Case 2) 2 ↔ 5 is included, but 5 ↔ 6 is not …

Spanning Trees

- **A spanning tree T** of an undirected graph **G** is …
  - **T** is a subgraph of **G**
  - **T** is a tree
  - **T** includes all the vertices of **G**

- How many spanning trees do exist in **G**?

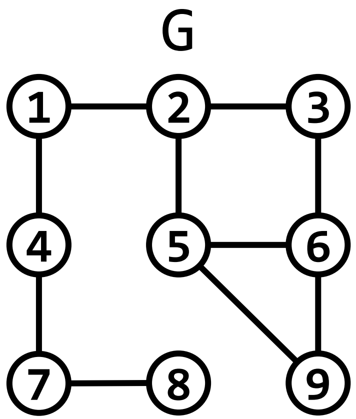


(Case 3) 2 ↔ 5 and 5 ↔ 6 are included …

Spanning Trees

- A spanning tree **T** of an undirected graph **G** is …
  - **T** is a subgraph of **G**
  - **T** is a tree
  - **T** includes all the vertices of **G**

- How many spanning trees do exist in **G**?

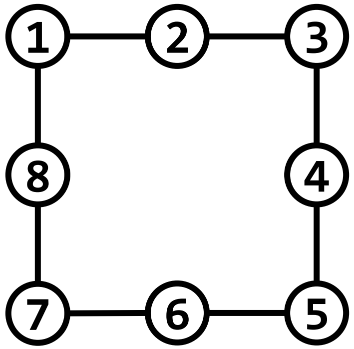


11 spanning trees exist in **G**

Spanning Trees

- **A spanning tree T** of an undirected graph **G** is …
  - **T** is a subgraph of **G**
  - **T** is a tree
  - **T** includes all the vertices of **G**

- How many spanning trees do exist in **G**?
  - If **G** is a simple cycle of **N** vertices, then # of spanning trees = **N**



Spanning Trees

- **A spanning tree T** of an undirected graph **G** is …
  - **T** is a subgraph of **G**
  - **T** is a tree
  - **T** includes all the vertices of **G**

- How many spanning trees do exist in **G**?
  - If **G** is a simple cycle of **N** vertices, then # of spanning trees = **N**
  - If **G** is a complete graph of **N** vertices, then # of spanning trees = $\mathbf{N^{N-2}}$
    - Known as Cayley's formula (proof is not covered in this lecture)

Spanning Trees

- **A spanning tree T** of an undirected graph **G** is …
  - **T** is a subgraph of **G**
  - **T** is a tree
  - **T** includes all the vertices of **G**

- How many spanning trees do exist in **G**?
  - If **G** is a simple cycle of **N** vertices, then # of spanning trees = **N**
  - If **G** is a complete graph of **N** vertices, then # of spanning trees = N^{N-2}
    - Known as Cayley's formula (proof is not covered in this lecture)
  - Can we compute the number of spanning trees in any graph?
    - Interestingly, it can be computed from the determinant of a submatrix of Laplacian matrix
      - Laplacian matrix **L** of a graph is defined by **D-A** (i.e., degree matrix – adjacent matrix)
      - Known as Kirchhoff's matrix tree theorem
    - This is also not covered in this lecture

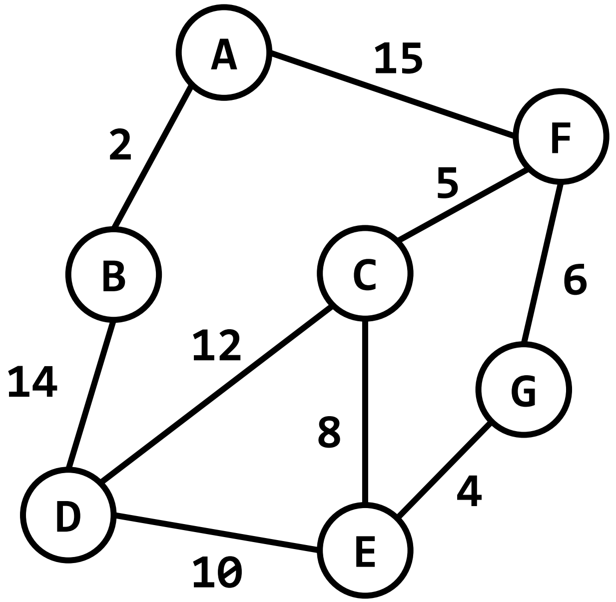
Minimum Spanning Trees

- Given a weighted graph **G**,
  - Let `w(e)` be the weight of the edge `e`
  - Let **T** be a spanning tree and `E(T)` be its edge set

  - The cost of spanning tree construction is the summation of all edge weights in **T**

$$\texttt{cost} = \sum_{\mathbf{e}\in E(T)} \mathbf{w}(\mathbf{e})$$

  - The **minimum spanning tree** is a spanning tree with minimum construction cost



Minimum Spanning Trees

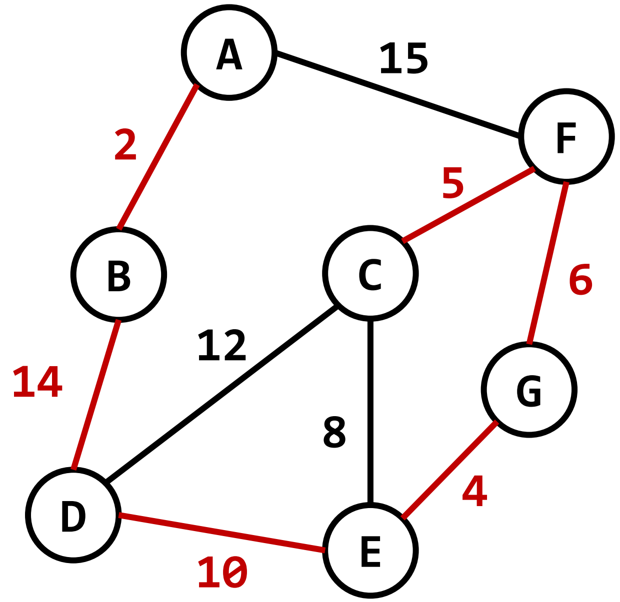
- Given a weighted graph **G**,
  - Let `w(e)` be the weight of the edge `e`
  - Let **T** be a spanning tree and `E(T)` be its edge set

  - The cost of spanning tree construction is the summation of all edge weights in **T**

$$\texttt{cost} = \sum_{\mathbf{e} \in E(T)} \mathbf{w}(\mathbf{e})$$

  - The **minimum spanning tree** is a spanning tree with minimum construction cost

```
cost = 2+14+10+4+6+5
     = 41
```

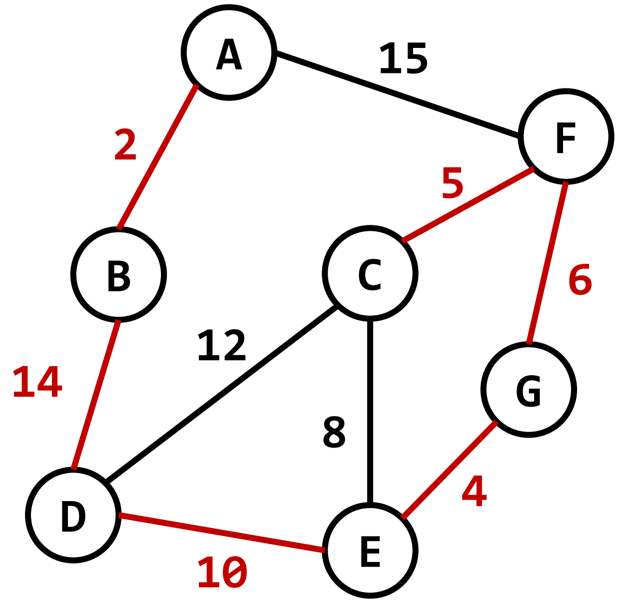


Minimum Spanning Trees

- Given a weighted graph **G**,
  - Let `w(e)` be the weight of the edge `e`
  - Let **T** be a spanning tree and `E(T)` be its edge set
  - The cost of spanning tree construction is the summation of all edge weights in **T**

$$\texttt{cost} = \sum_{\mathbf{e} \in E(T)} \mathbf{w}(\mathbf{e})$$

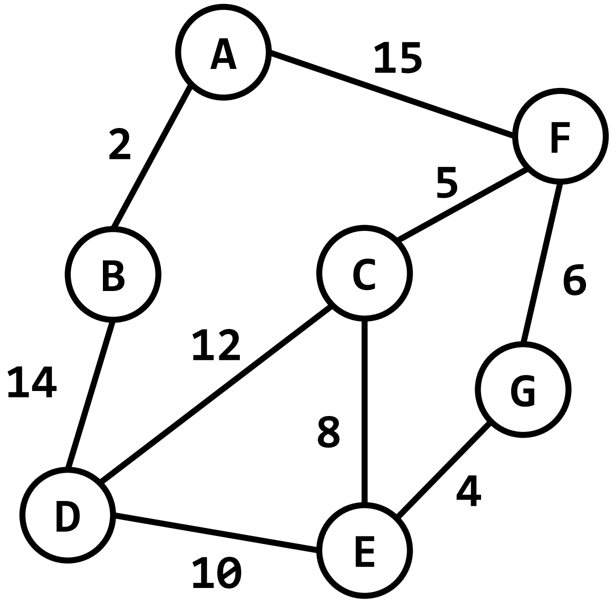
  - The **minimum spanning tree** is a spanning tree with minimum construction cost



(Q) How to find the minimum spanning tree?
  - Prim's Algorithm (vertex-based)
  - Kruskal's Algorithm (edge-based)

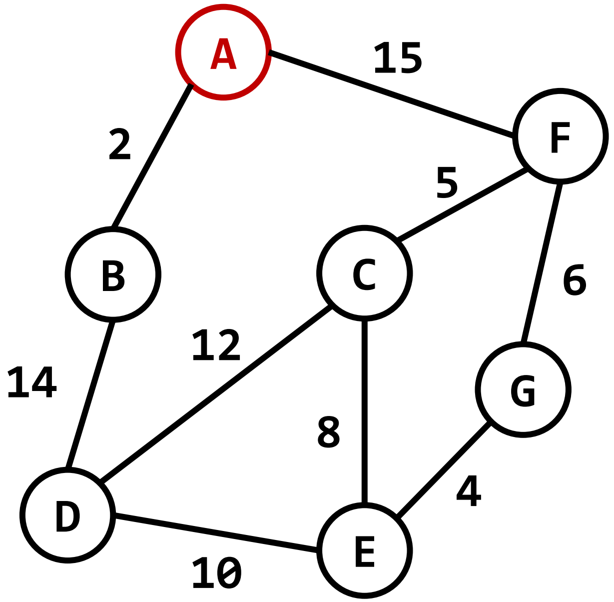
Prim's Algorithm

- The key idea is to add vertex one by one into the vertex set
  1. Start from a single-vertex subgraph `T=(U,F)` where `U = { A }` and `F = Ø`
  2. Pick the vertex of the minimum **addition cost** not yet included in `T`
  3. **Add the vertex** into `T`
  4. Repeat 2 & 3 steps



Prim's Algorithm

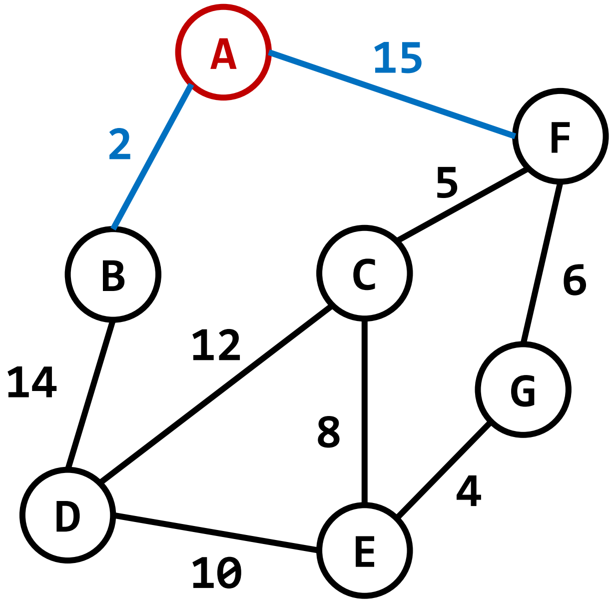
- The key idea is to add vertex one by one into the vertex set
  1. Start from a single-vertex subgraph `T=(U,F)` where `U = { A }` and `F = Ø`
  2. Pick the vertex of the minimum **addition cost** not yet included in `T`
  3. **Add the vertex** into `T`
  4. Repeat 2 & 3 steps



U = { A }

Prim's Algorithm

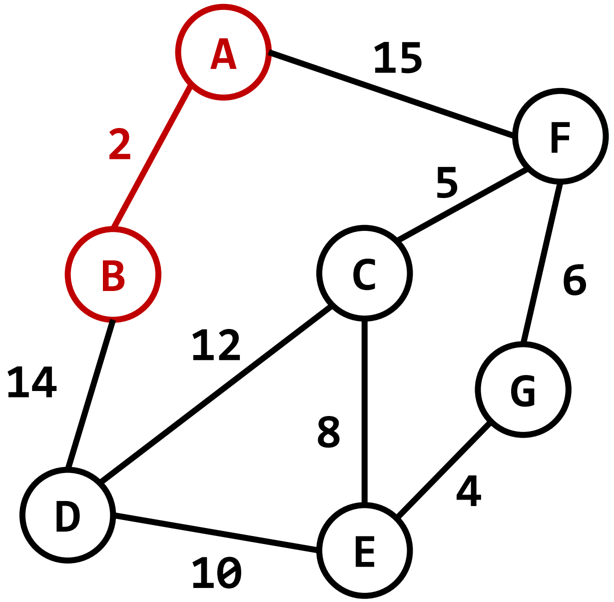
- The key idea is to add vertex one by one into the vertex set
  1. Start from a single-vertex subgraph `T=(U,F)` where `U = { A }` and `F = Ø`
  2. Pick the vertex of the minimum **addition cost** not yet included in `T`
  3. **Add the vertex** into `T`
  4. Repeat 2 & 3 steps



U = { A }

Prim's Algorithm

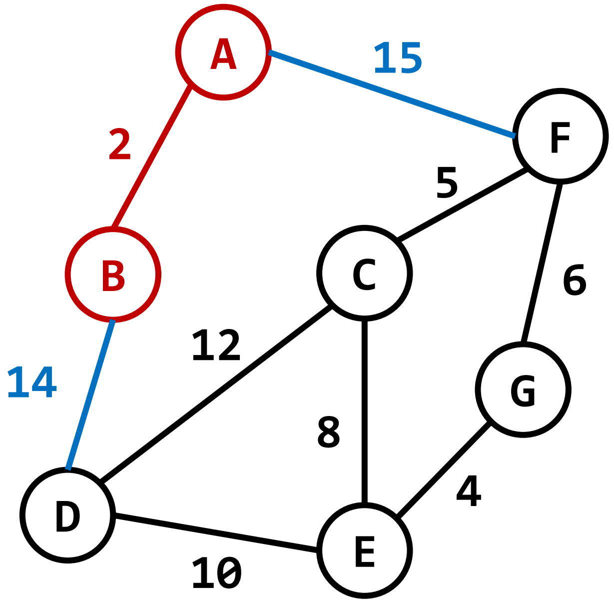
- The key idea is to add vertex one by one into the vertex set
  1. Start from a single-vertex subgraph `T=(U,F)` where `U` = { A } and `F` = Ø
  2. Pick the vertex of the minimum **addition cost** not yet included in `T`
  3. **Add the vertex** into `T`
  4. Repeat 2 & 3 steps



U = { A B }

Prim's Algorithm

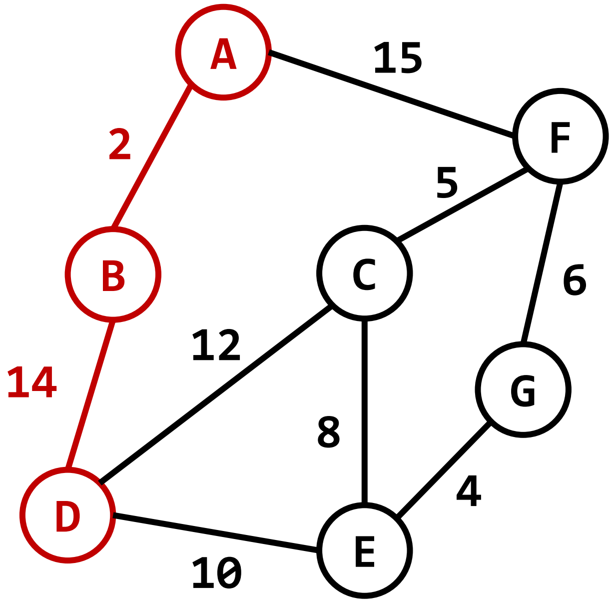
- The key idea is to add vertex one by one into the vertex set
  1. Start from a single-vertex subgraph `T=(U,F)` where `U = { A }` and `F = Ø`
  2. Pick the vertex of the minimum **addition cost** not yet included in `T`
  3. **Add the vertex** into `T`
  4. Repeat 2 & 3 steps



U = { A B }

Prim's Algorithm

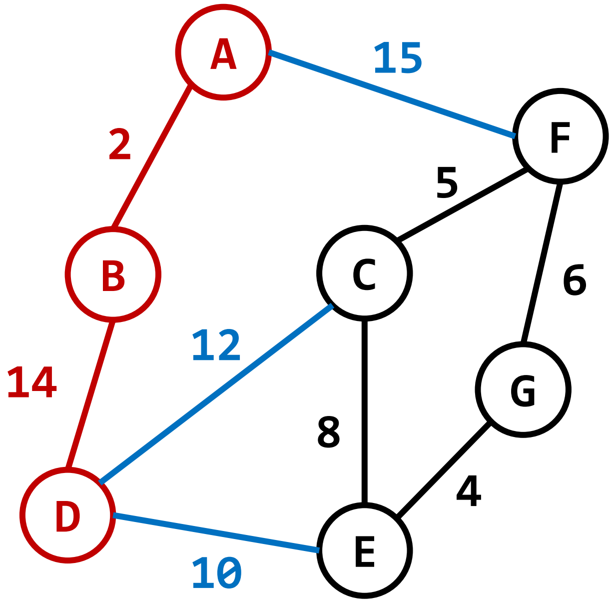
- The key idea is to add vertex one by one into the vertex set
  1. Start from a single-vertex subgraph `T=(U,F)` where `U = { A }` and `F = Ø`
  2. Pick the vertex of the minimum **addition cost** not yet included in `T`
  3. **Add the vertex** into `T`
  4. Repeat 2 & 3 steps



U = { A B D }

Prim's Algorithm

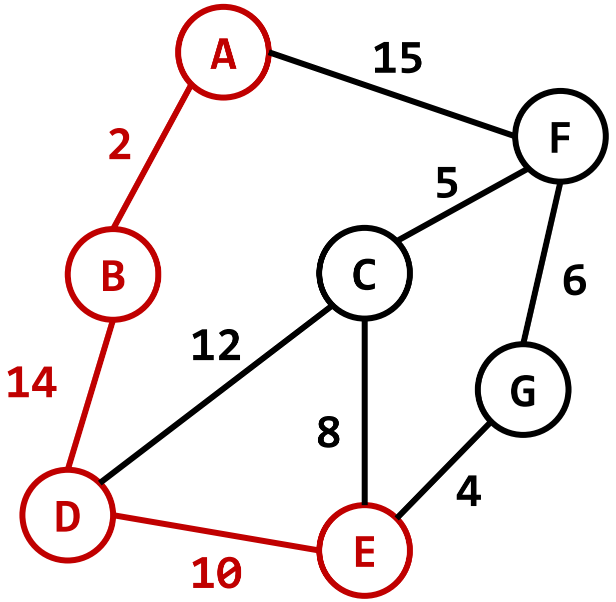
- The key idea is to add vertex one by one into the vertex set
  1. Start from a single-vertex subgraph `T=(U,F)` where `U` = { A } and `F` = Ø
  2. Pick the vertex of the minimum addition cost not yet included in `T`
  3. **Add the vertex** into `T`
  4. Repeat 2 & 3 steps



U = { A B D }

Prim's Algorithm

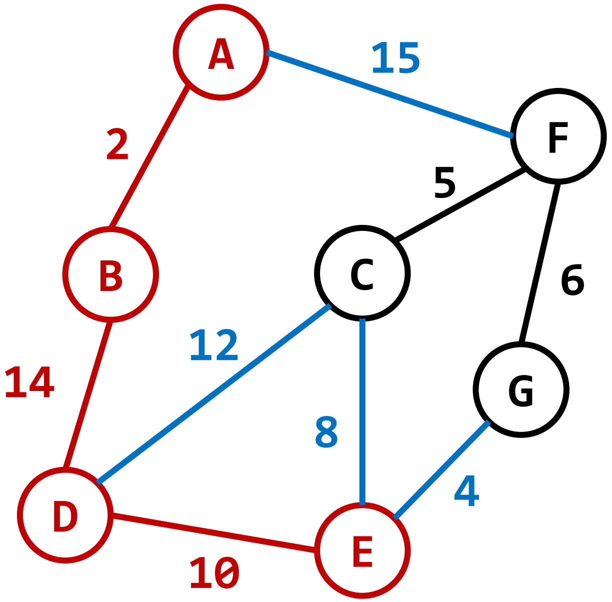
- The key idea is to add vertex one by one into the vertex set
  1. Start from a single-vertex subgraph `T=(U,F)` where `U = { A }` and `F = Ø`
  2. Pick the vertex of the minimum **addition cost** not yet included in `T`
  3. **Add the vertex** into `T`
  4. Repeat 2 & 3 steps



U = { A B D E }

Prim's Algorithm

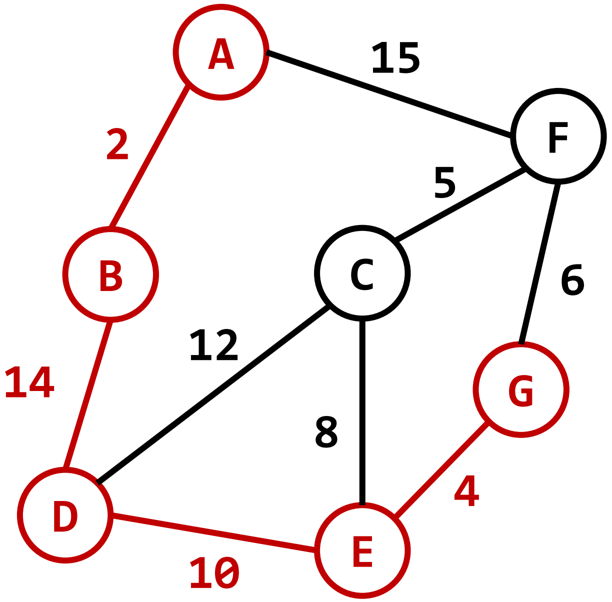
- The key idea is to add vertex one by one into the vertex set
  1. Start from a single-vertex subgraph `T=(U,F)` where `U = { A }` and `F = Ø`
  2. Pick the vertex of the minimum addition cost not yet included in `T`
  3. **Add the vertex** into `T`
  4. Repeat 2 & 3 steps



U = { A B D E }

Prim's Algorithm

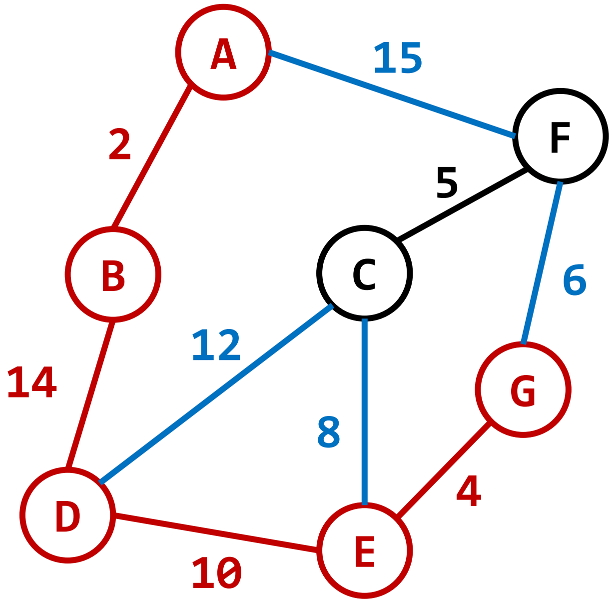
- The key idea is to add vertex one by one into the vertex set
  1. Start from a single-vertex subgraph `T=(U,F)` where `U = { A }` and `F = Ø`
  2. Pick the vertex of the minimum **addition cost** not yet included in `T`
  3. **Add the vertex** into `T`
  4. Repeat 2 & 3 steps



U = { A B D E G }

Prim's Algorithm

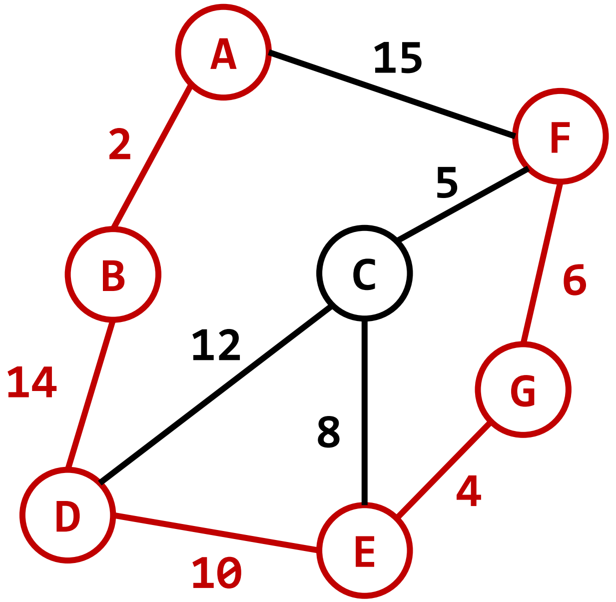
- The key idea is to add vertex one by one into the vertex set
  1. Start from a single-vertex subgraph `T=(U,F)` where `U` = { A } and `F` = Ø
  2. Pick the vertex of the minimum addition cost not yet included in `T`
  3. **Add the vertex** into `T`
  4. Repeat 2 & 3 steps



U = { A B D E G }

Prim's Algorithm

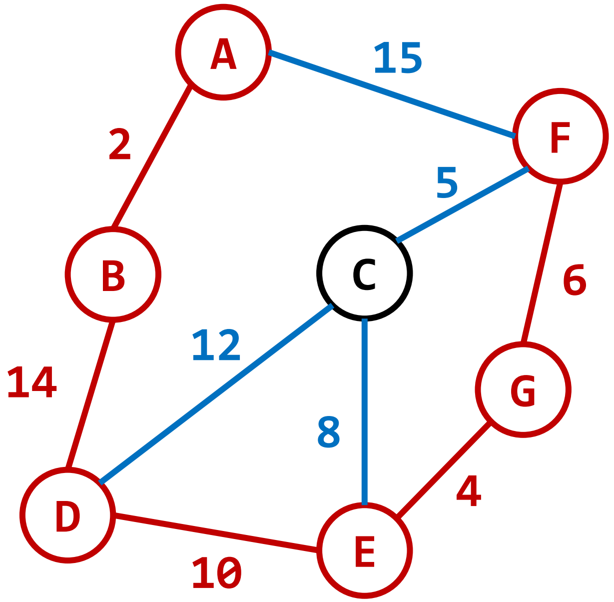
- The key idea is to add vertex one by one into the vertex set
  1. Start from a single-vertex subgraph T=(U,F) where U = { A } and F = Ø
  2. Pick the vertex of the minimum **addition cost** not yet included in T
  3. **Add the vertex** into T
  4. Repeat 2 & 3 steps



U = { A B D E G F }

Prim's Algorithm

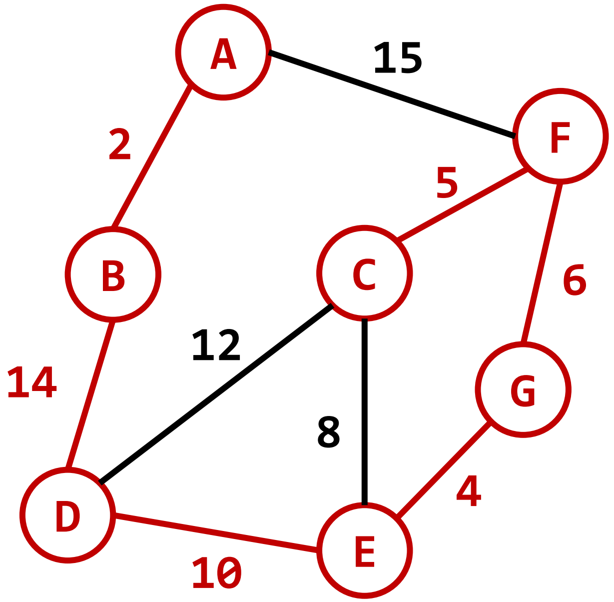
- The key idea is to add vertex one by one into the vertex set
  1. Start from a single-vertex subgraph `T=(U,F)` where `U = { A }` and `F = Ø`
  2. Pick the vertex of the minimum addition cost not yet included in `T`
  3. **Add the vertex** into `T`
  4. Repeat 2 & 3 steps



U = { A B D E G F }

Prim's Algorithm

- The key idea is to add vertex one by one into the vertex set
  1. Start from a single-vertex subgraph `T=(U,F)` where `U = { A }` and `F = Ø`
  2. Pick the vertex of the minimum **addition cost** not yet included in `T`
  3. **Add the vertex** into `T`
  4. Repeat 2 & 3 steps

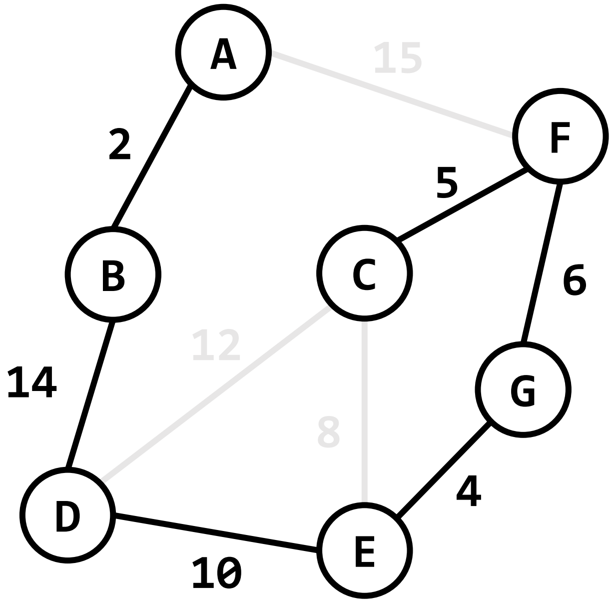


U = { A B D E G F C }

Prim's Algorithm

- The key idea is to add vertex one by one into the vertex set
  1. Start from a single-vertex subgraph `T=(U,F)` where `U = { A }` and `F = Ø`
  2. Pick the vertex of the minimum **addition cost** not yet included in `T`
  3. **Add the vertex** into `T`
  4. Repeat 2 & 3 steps

The resultant graph is the minimum spanning tree!



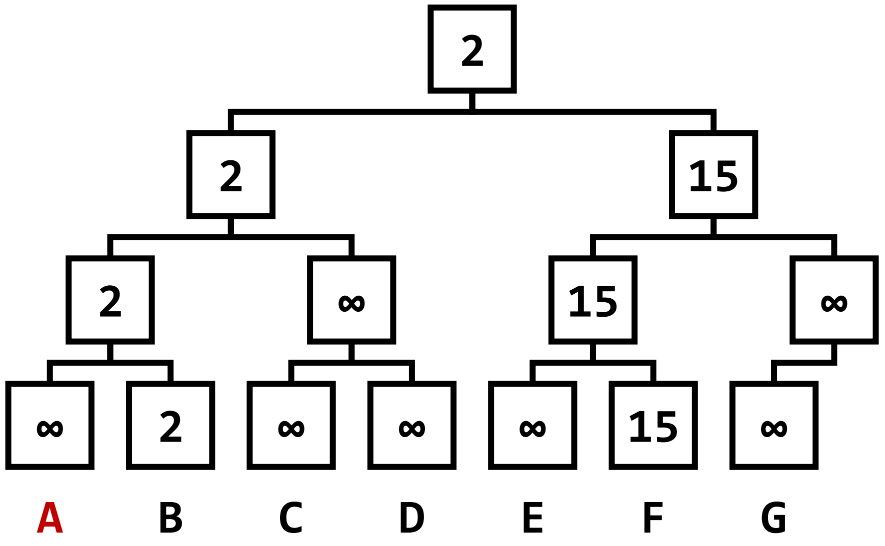
Prim's Algorithm

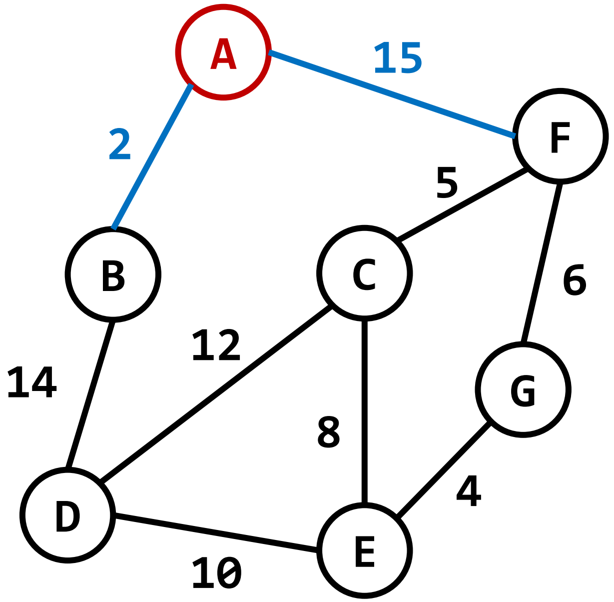
- The key idea is to add vertex one by one into the vertex set
  1. Start from a single-vertex subgraph `T=(U,F)` where `U = { A }` and `F = Ø`
  2. Pick the vertex of the minimum **addition cost** not yet included in `T`
  3. **Add the vertex** into `T`
  4. Repeat 2 & 3 steps

- How to implement this algorithm efficiently?
  - Maintain **the addition cost array** for N vertices
    - Set ∞ (e.g., a very large value) for vertices (a) already included in T or (b) cannot be added
  - Find the vertex of the minimum addition cost **using Segment Tree**
    - It can efficiently find the minimum cost among all vertices
    - It can efficiently update the cost

Prim's Algorithm

- The key idea is to add vertex one by one into the vertex set
  1. Start from a single-vertex subgraph T=(U,F) where U = { A } and F = Ø
  2. Pick the vertex of the minimum **addition cost** not yet included in T
  3. **Add the vertex** into T
  4. Repeat 2 & 3 steps

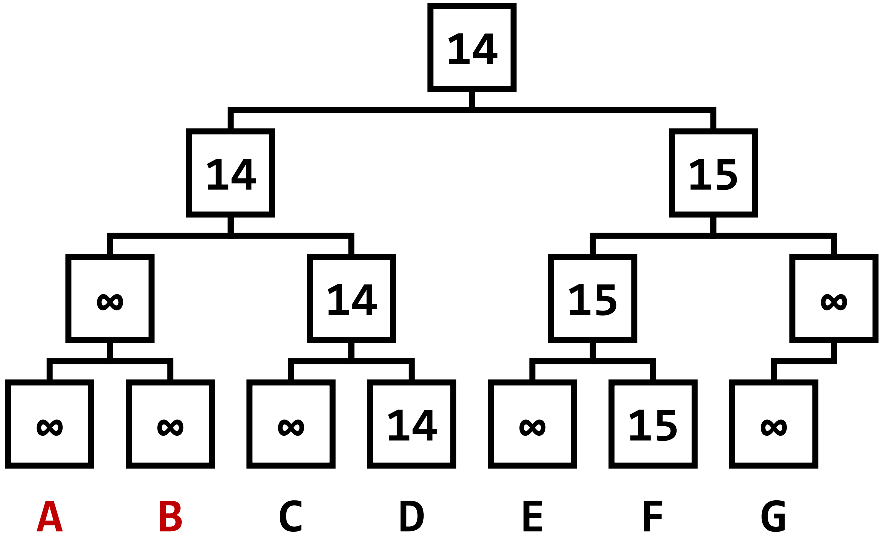


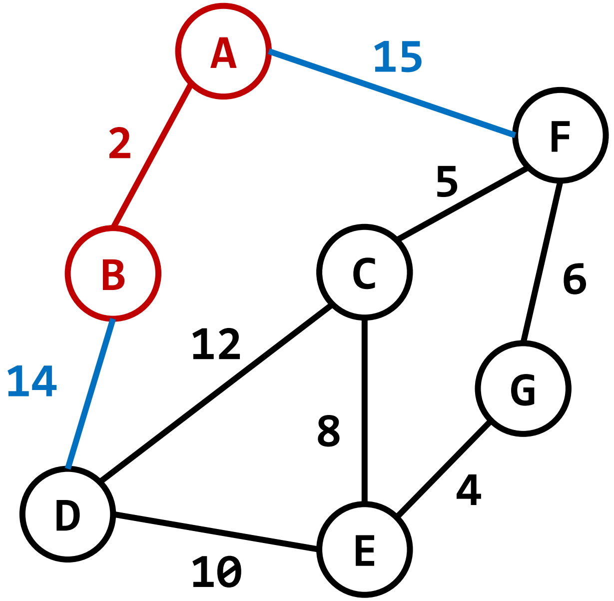


U = { A }

Prim's Algorithm

- The key idea is to add vertex one by one into the vertex set
  1. Start from a single-vertex subgraph `T=(U,F)` where `U = { A }` and `F = Ø`
  2. Pick the vertex of the minimum **addition cost** not yet included in `T`
  3. **Add the vertex** into `T`
  4. Repeat 2 & 3 steps

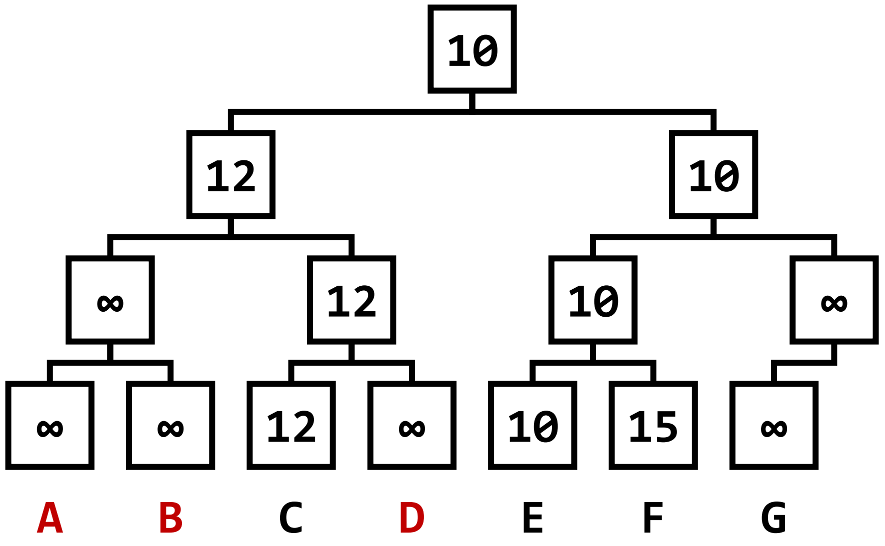


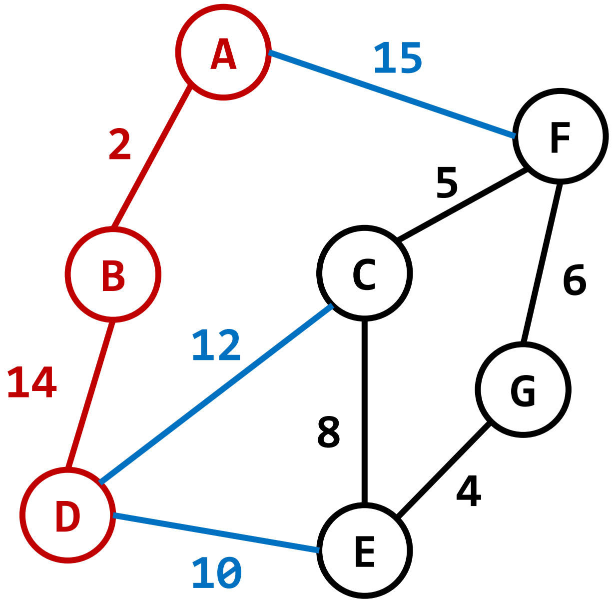


U = { A B }

Prim's Algorithm

- The key idea is to add vertex one by one into the vertex set
  1. Start from a single-vertex subgraph `T=(U,F)` where `U = { A }` and `F = Ø`
  2. Pick the vertex of the minimum **addition cost** not yet included in `T`
  3. **Add the vertex** into `T`
  4. Repeat 2 & 3 steps

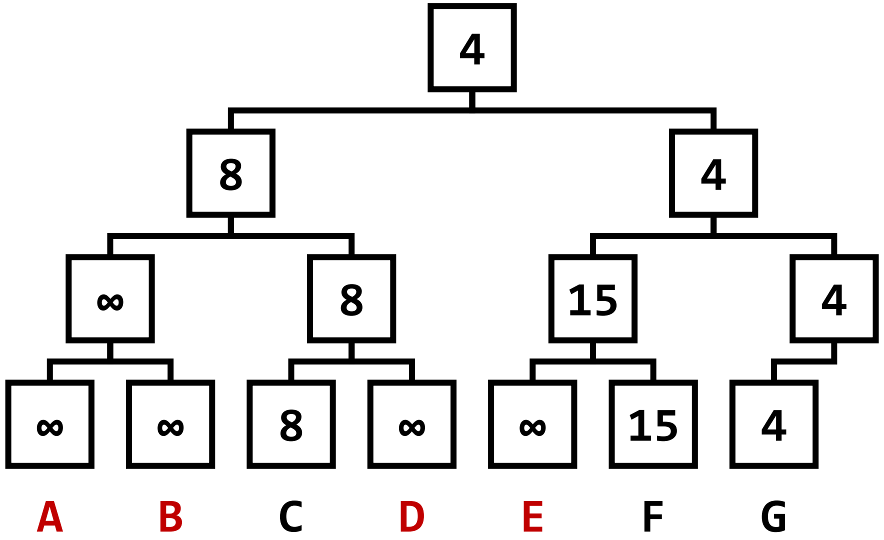


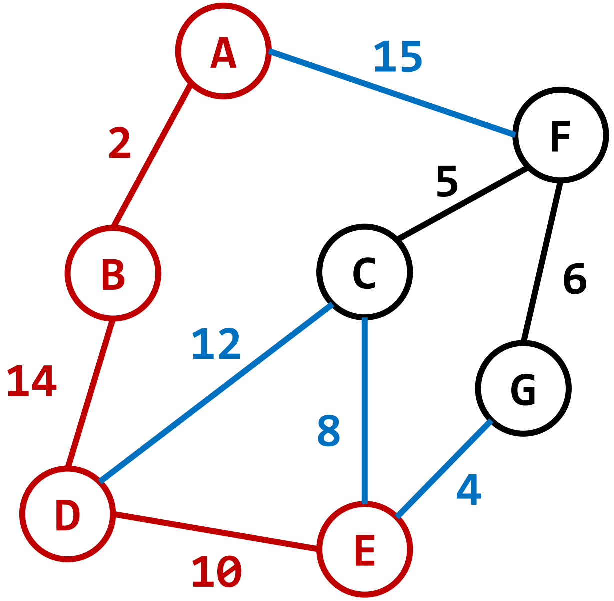


U = { A B D }

Prim's Algorithm

- The key idea is to add vertex one by one into the vertex set
  1. Start from a single-vertex subgraph `T=(U,F)` where `U = { A }` and `F = Ø`
  2. Pick the vertex of the minimum **addition cost** not yet included in `T`
  3. **Add the vertex** into `T`
  4. Repeat 2 & 3 steps

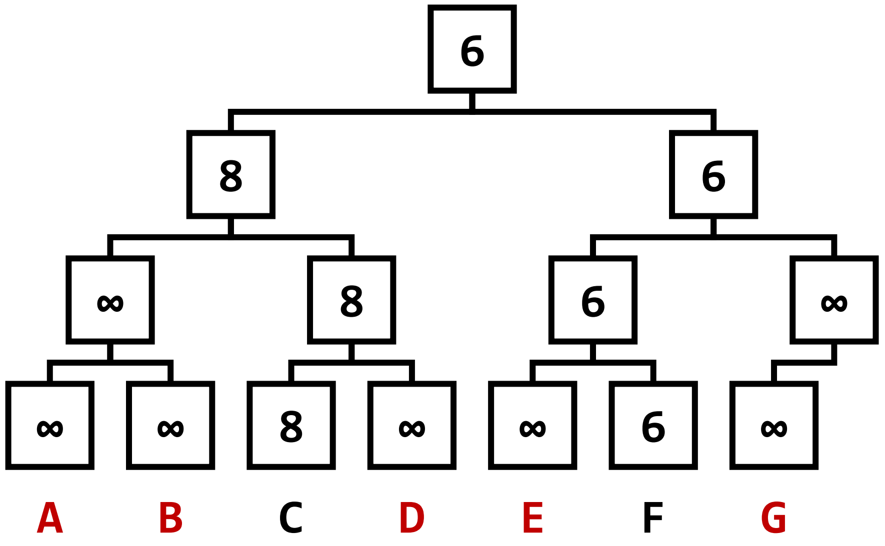


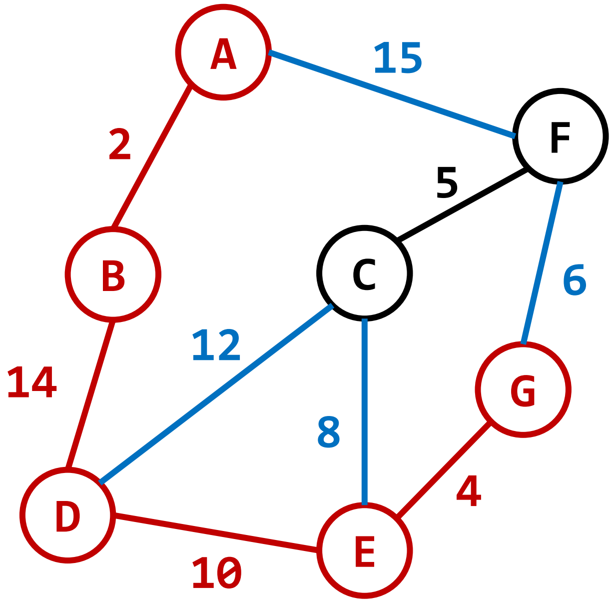


U = { A B D E }

Prim's Algorithm

- The key idea is to add vertex one by one into the vertex set
  1. Start from a single-vertex subgraph `T=(U,F)` where `U = { A }` and `F = Ø`
  2. Pick the vertex of the minimum **addition cost** not yet included in `T`
  3. **Add the vertex** into `T`
  4. Repeat 2 & 3 steps

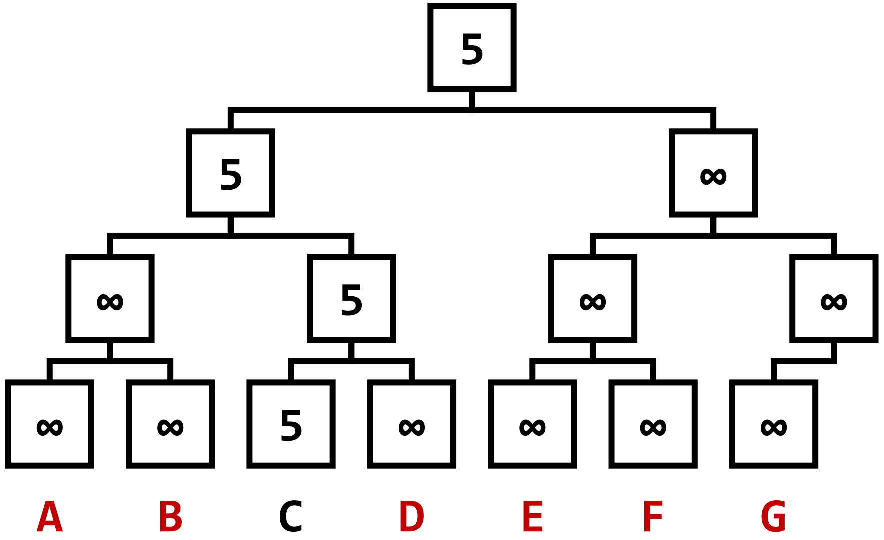


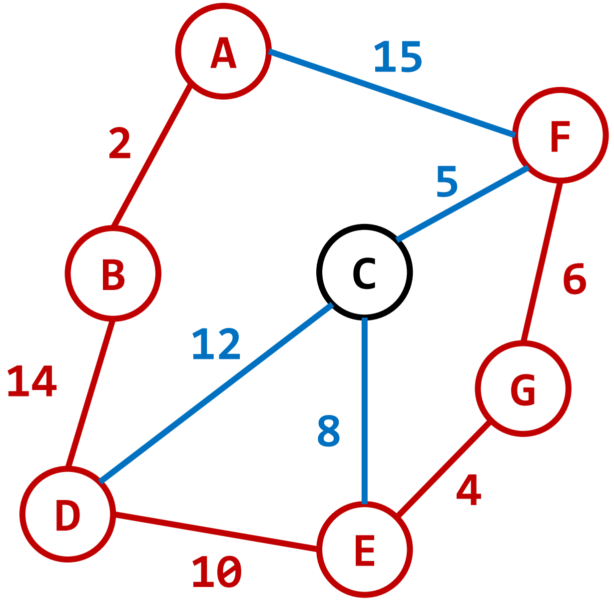


U = { A B D E G }

Prim's Algorithm

- The key idea is to add vertex one by one into the vertex set
  1. Start from a single-vertex subgraph `T=(U,F)` where `U = { A }` and `F = Ø`
  2. Pick the vertex of the minimum **addition cost** not yet included in `T`
  3. **Add the vertex** into `T`
  4. Repeat 2 & 3 steps

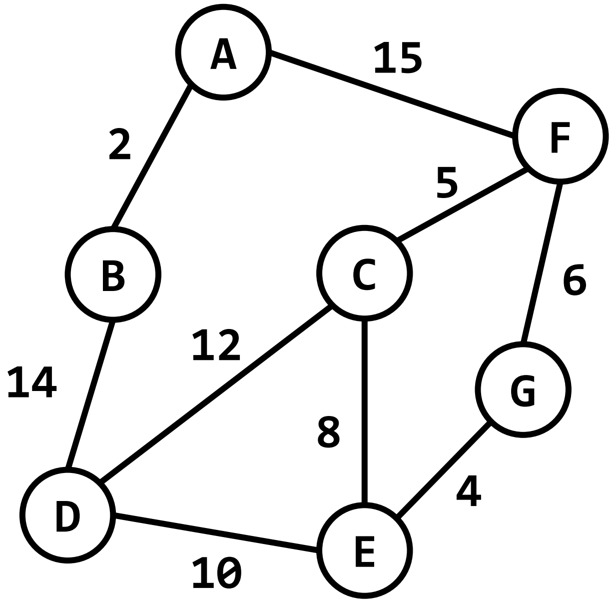




U = { A B D E G F }

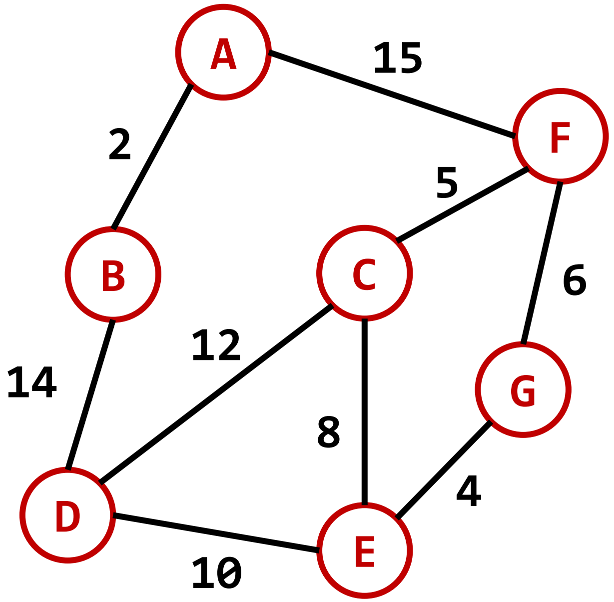
Kruskal's Algorithm

- The key idea is to add edge one by one into the edge set
  1. Start from a zero-edge subgraph `T=(V,F)` where `V` = { all vertices } and `F` = ∅
  2. Pick the edge of the minimum **addition cost** not form a cycle in `T`
  3. **Add the edge** into `T`
  4. Repeat 2 & 3 steps



Kruskal's Algorithm

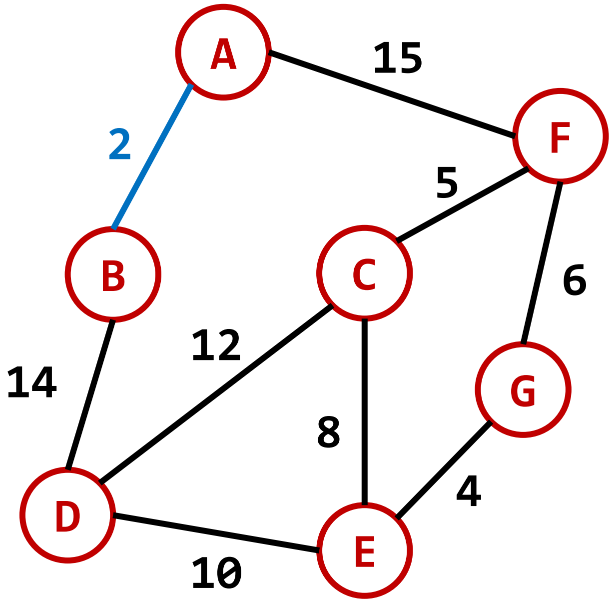
- The key idea is to add edge one by one into the edge set
  1. Start from a zero-edge subgraph T=(V,F) where V = { all vertices } and F = ∅
  2. Pick the edge of the minimum **addition cost** not form a cycle in T
  3. **Add the edge** into T
  4. Repeat 2 & 3 steps



F = { }

Kruskal's Algorithm

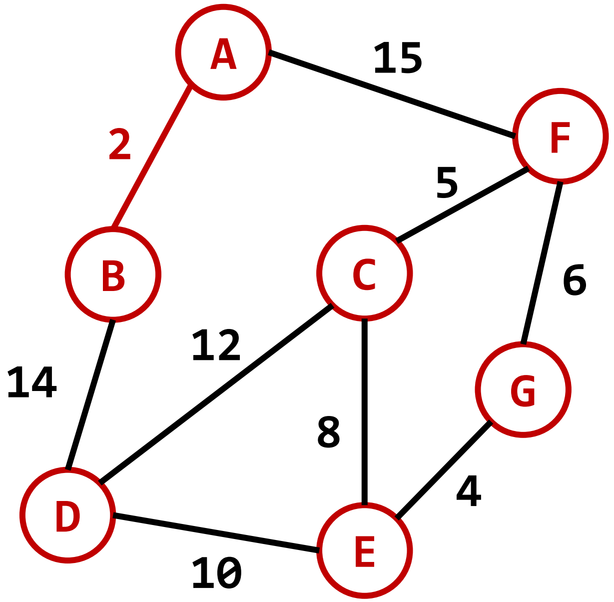
- The key idea is to add edge one by one into the edge set
  1. Start from a zero-edge subgraph T=(V,F) where V = { all vertices } and F = Ø
  2. Pick the edge of the minimum addition cost not form a cycle in T
  3. **Add the edge** into T
  4. Repeat 2 & 3 steps



F = { }

Kruskal's Algorithm

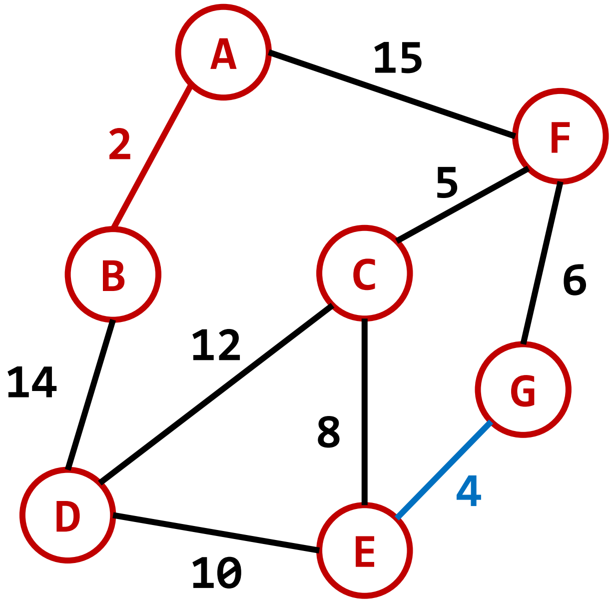
- The key idea is to add edge one by one into the edge set
  1. Start from a zero-edge subgraph `T=(V,F)` where `V` = { all vertices } and `F` = ∅
  2. Pick the edge of the minimum **addition cost** not form a cycle in `T`
  3. **Add the edge** into `T`
  4. Repeat 2 & 3 steps



F = { AB }

Kruskal's Algorithm

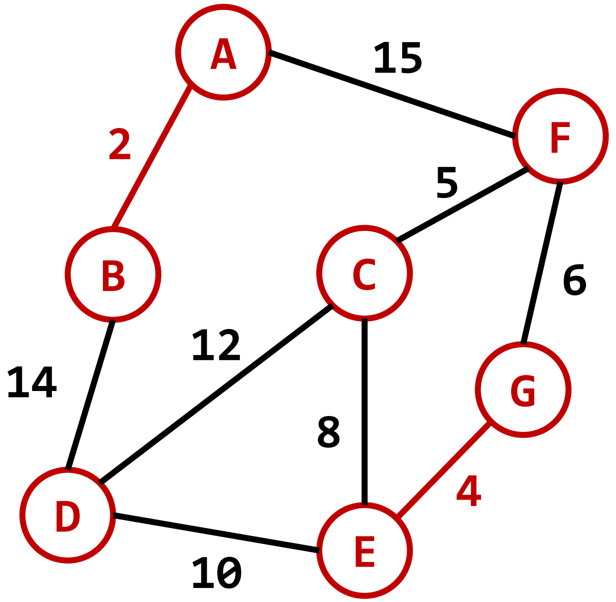
- The key idea is to add edge one by one into the edge set
  1. Start from a zero-edge subgraph `T=(V,F)` where `V` = { all vertices } and `F` = ∅
  2. Pick the edge of the minimum **addition cost** not form a cycle in `T`
  3. **Add the edge** into `T`
  4. Repeat 2 & 3 steps



F = { AB }

Kruskal's Algorithm

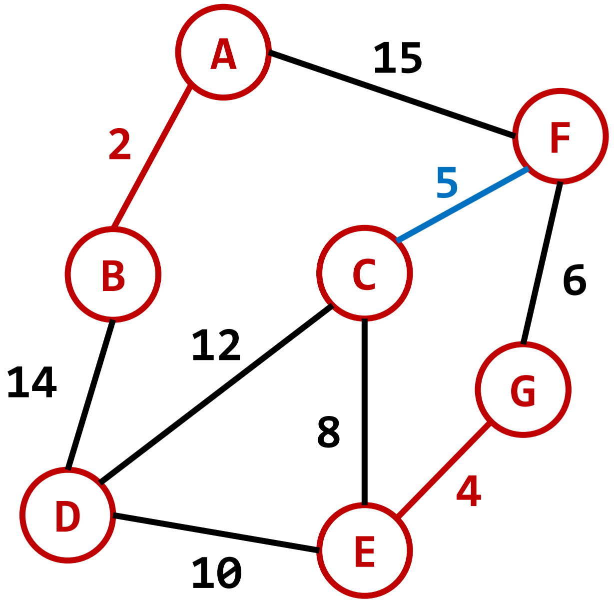
- The key idea is to add edge one by one into the edge set
  1. Start from a zero-edge subgraph `T=(V,F)` where `V` = { all vertices } and `F` = ∅
  2. Pick the edge of the minimum **addition cost** not form a cycle in `T`
  3. **Add the edge** into `T`
  4. Repeat 2 & 3 steps



F = { AB EG }

Kruskal's Algorithm

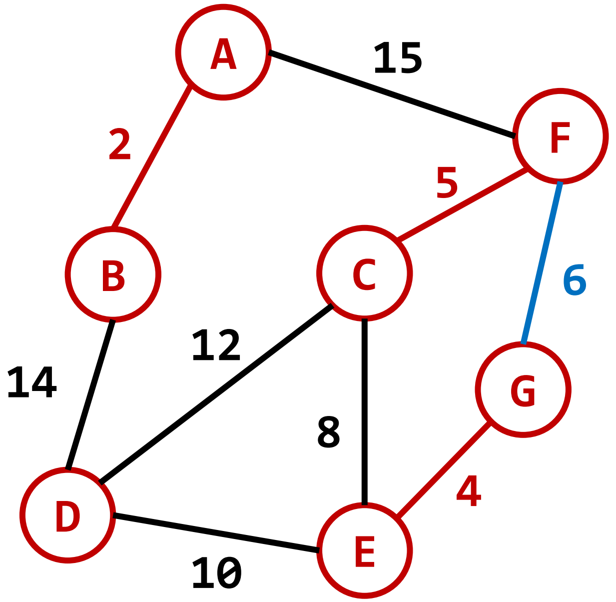
- The key idea is to add edge one by one into the edge set
  1. Start from a zero-edge subgraph `T=(V,F)` where `V` = { all vertices } and `F` = ∅
  2. Pick the edge of the minimum **addition cost** not form a cycle in `T`
  3. **Add the edge** into `T`
  4. Repeat 2 & 3 steps



F = { AB EG }

Kruskal's Algorithm

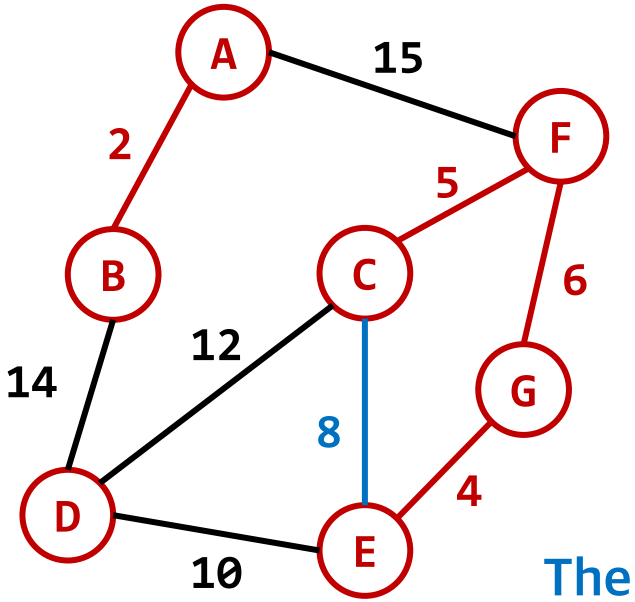
- The key idea is to add edge one by one into the edge set
  1. Start from a zero-edge subgraph `T=(V,F)` where `V` = { all vertices } and `F` = ∅
  2. Pick the edge of the minimum **addition cost** not form a cycle in `T`
  3. **Add the edge** into `T`
  4. Repeat 2 & 3 steps



F = { AB EG CF }

Kruskal's Algorithm

- The key idea is to add edge one by one into the edge set
  1. Start from a zero-edge subgraph `T=(V,F)` where `V` = { all vertices } and `F` = Ø
  2. Pick the edge of the minimum **addition cost** not form a cycle in `T`
  3. **Add the edge** into `T`
  4. Repeat 2 & 3 steps

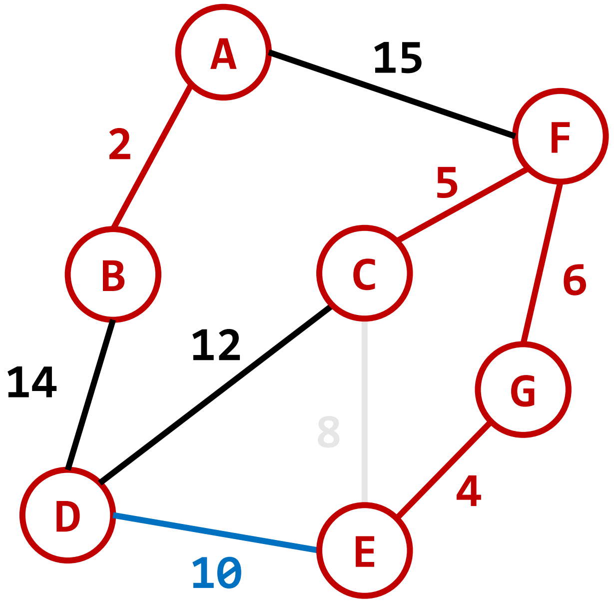


F = { AB EG CF FG }

The edge forms the cycle **CEGF**

Kruskal's Algorithm

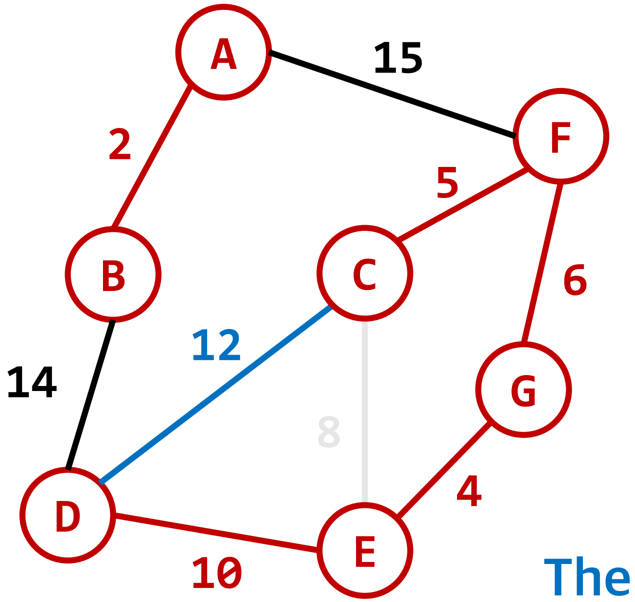
- The key idea is to add edge one by one into the edge set
  1. Start from a zero-edge subgraph `T=(V,F)` where `V` = { all vertices } and `F` = ∅
  2. Pick the edge of the minimum **addition cost** not form a cycle in `T`
  3. **Add the edge** into `T`
  4. Repeat 2 & 3 steps



F = { AB EG CF FG }

Kruskal's Algorithm

- The key idea is to add edge one by one into the edge set
  1. Start from a zero-edge subgraph `T=(V,F)` where `V` = { all vertices } and `F` = ∅
  2. Pick the edge of the minimum **addition cost** not form a cycle in `T`
  3. **Add the edge** into `T`
  4. Repeat 2 & 3 steps

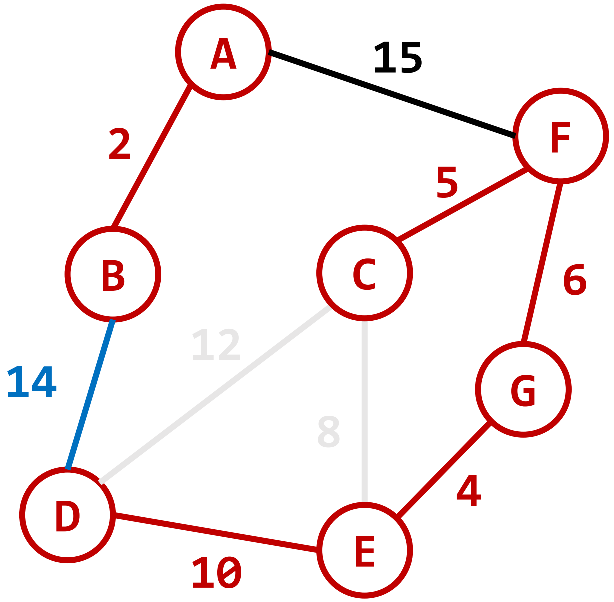


F = { AB EG CF FG DE }

The edge forms the cycle **CDEGF**

Kruskal's Algorithm

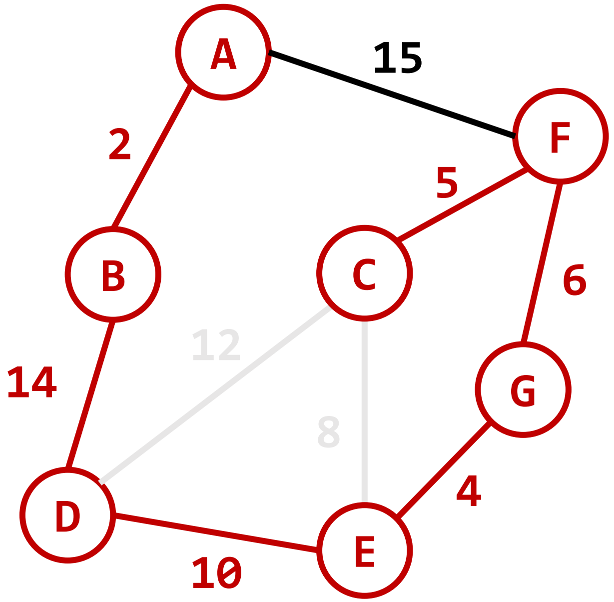
- The key idea is to add edge one by one into the edge set
  1. Start from a zero-edge subgraph `T=(V,F)` where `V` = { all vertices } and `F` = ∅
  2. Pick the edge of the minimum **addition cost** not form a cycle in `T`
  3. **Add the edge** into `T`
  4. Repeat 2 & 3 steps



F = { AB EG CF FG DE }

Kruskal's Algorithm

- The key idea is to add edge one by one into the edge set
  1. Start from a zero-edge subgraph T=(V,F) where V = { all vertices } and F = Ø
  2. Pick the edge of the minimum **addition cost** not form a cycle in T
  3. **Add the edge** into T
  4. Repeat 2 & 3 steps

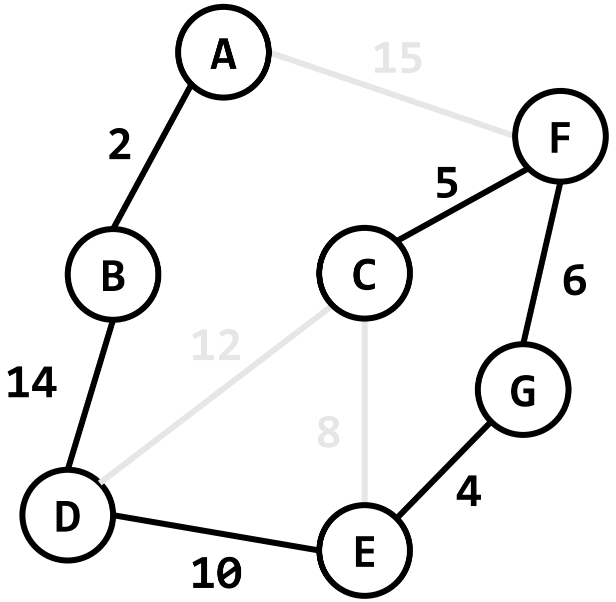


F = { AB EG CF FG DE BD }

Kruskal's Algorithm

- The key idea is to add edge one by one into the edge set
  1. Start from a zero-edge subgraph `T=(V,F)` where `V` = { all vertices } and `F` = ∅
  2. Pick the edge of the minimum **addition cost** not form a cycle in `T`
  3. **Add the edge** into `T`
  4. Repeat 2 & 3 steps

The resultant graph is the minimum spanning tree!



Kruskal's Algorithm

- The key idea is to add edge one by one into the edge set
    1. Start from a zero-edge subgraph T=(V,F) where V = { all vertices } and F = Ø
    2. Pick the edge of the minimum **addition cost** not form a cycle in T
    3. **Add the edge** into T
    4. Repeat 2 & 3 steps

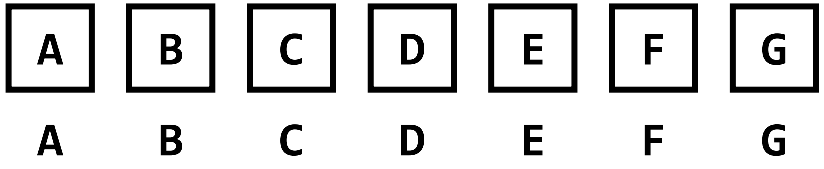
- How to implement this algorithm efficiently?
    - Sorting edges by weights in the increasing order
    - Use the disjoint-set structure to check an edge addition forms a cycle
        - The structure is also called union-find
        - The structure is not covered in the final exam, but I'll explain its high-level concept in next slides

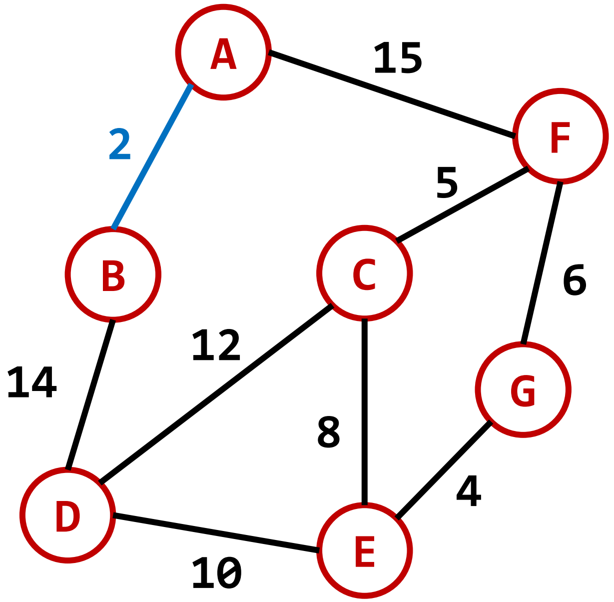
Kruskal's Algorithm

- The key idea is to add edge one by one into the edge set
  1. Start from a zero-edge subgraph T=(V,F) where V = { all vertices } and F = Ø
  2. Pick the edge of the minimum **addition cost** not form a cycle in T
  3. **Add the edge** into T
  4. Repeat 2 & 3 steps

If **the edge** connects two disjoint sets, merge them
Otherwise, skip the edge

X → **Y** means
X belongs to the set including **Y**





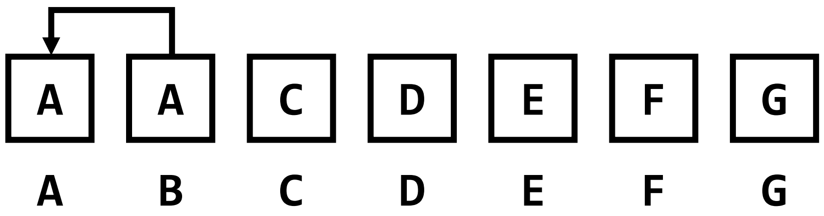
Disjoint sets
{ A }
{ B }
{ C }
{ D }
{ E }
{ F }
{ G }

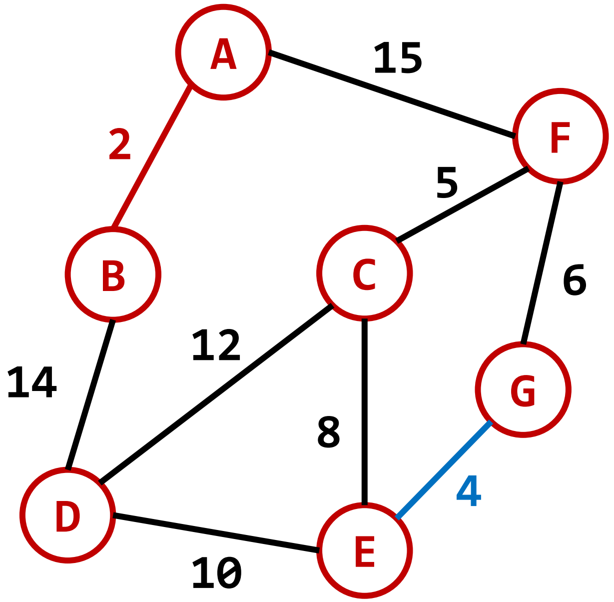
Kruskal's Algorithm

- The key idea is to add edge one by one into the edge set
  1. Start from a zero-edge subgraph `T=(V,F)` where `V` = { all vertices } and `F` = ∅
  2. Pick the edge of the minimum **addition cost** not form a cycle in `T`
  3. **Add the edge** into `T`
  4. Repeat 2 & 3 steps

If **the edge** connects two disjoint sets, merge them
Otherwise, skip the edge

X → **Y** means
X belongs to the set including **Y**





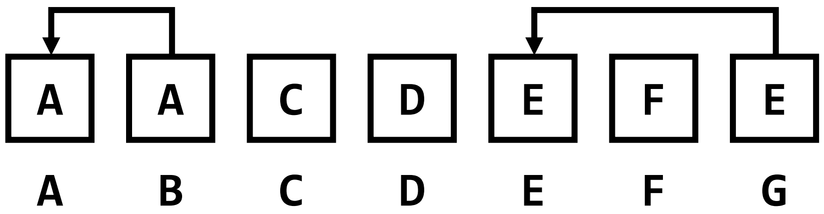
Disjoint sets
{ A B }
{ C }
{ D }
{ E }
{ F }
{ G }

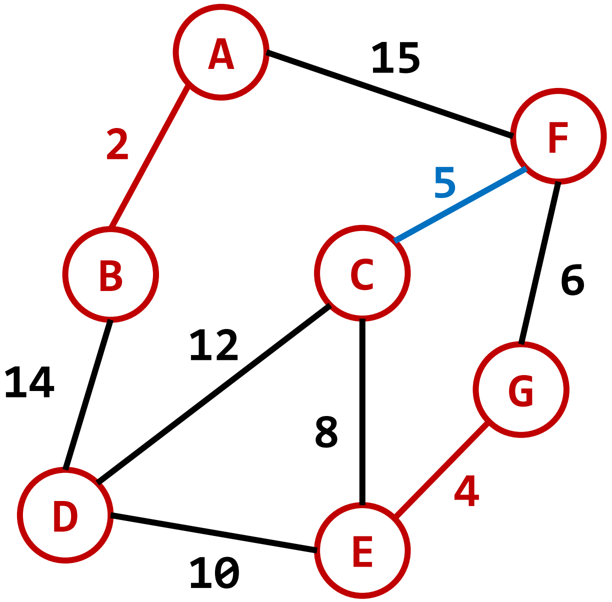
Kruskal's Algorithm

- The key idea is to add edge one by one into the edge set
  1. Start from a zero-edge subgraph `T=(V,F)` where `V` = { all vertices } and `F` = ∅
  2. Pick the edge of the minimum **addition cost** not form a cycle in `T`
  3. **Add the edge** into `T`
  4. Repeat 2 & 3 steps

If **the edge** connects two disjoint sets, merge them
Otherwise, skip the edge

X → **Y** means
X belongs to the set including **Y**





Disjoint sets

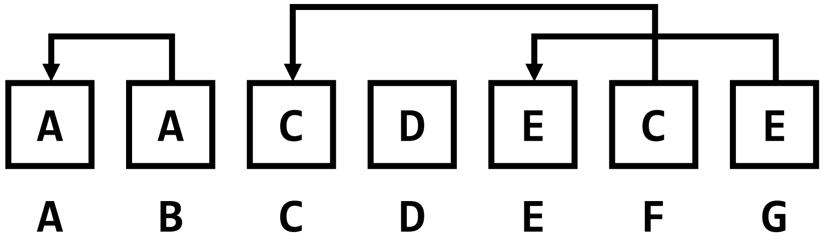
{ A B }
{ C }
{ D }
{ E G }
{ F }

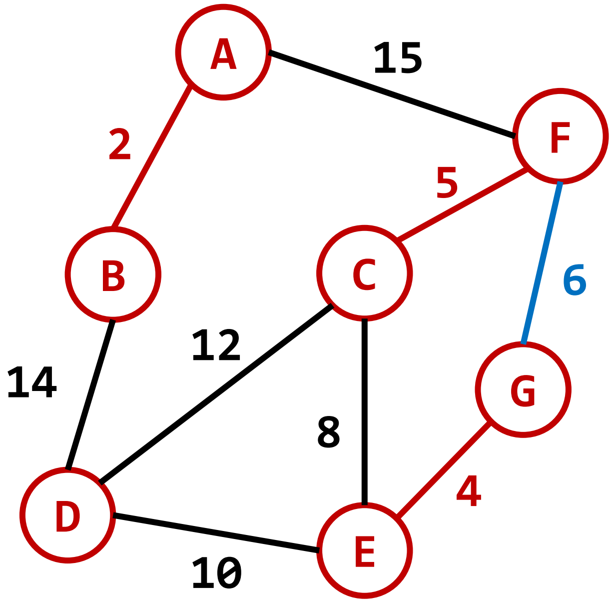
Kruskal's Algorithm

- The key idea is to add edge one by one into the edge set
  1. Start from a zero-edge subgraph `T=(V,F)` where V = { all vertices } and F = Ø
  2. Pick the edge of the minimum **addition cost** not form a cycle in `T`
  3. **Add the edge** into `T`
  4. Repeat 2 & 3 steps

If **the edge** connects two disjoint sets, merge them
Otherwise, skip the edge

X → **Y** means
X belongs to the set including **Y**





Disjoint sets

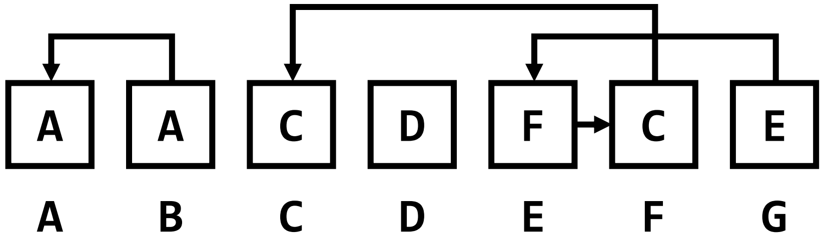
{ A B }
{ C F }
{ D }
{ E G }

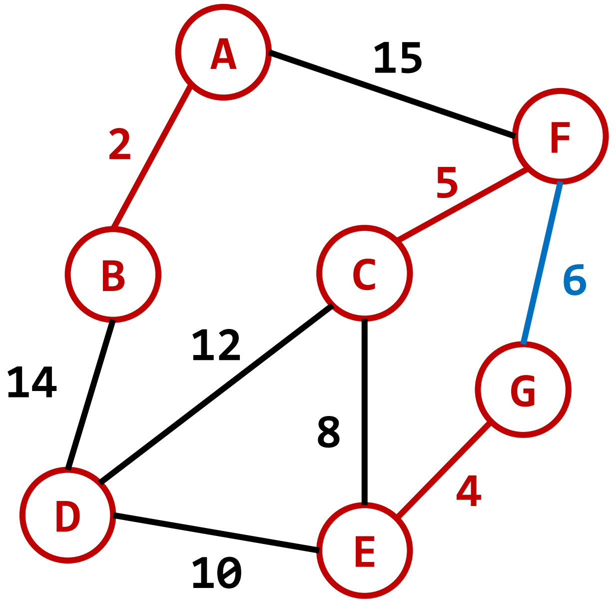
Kruskal's Algorithm

- The key idea is to add edge one by one into the edge set
  1. Start from a zero-edge subgraph `T=(V,F)` where `V` = { all vertices } and `F` = ∅
  2. Pick the edge of the minimum **addition cost** not form a cycle in `T`
  3. **Add the edge** into `T`
  4. Repeat 2 & 3 steps

If **the edge** connects two disjoint sets, merge them
Otherwise, skip the edge

X → **Y** means
X belongs to the set including **Y**





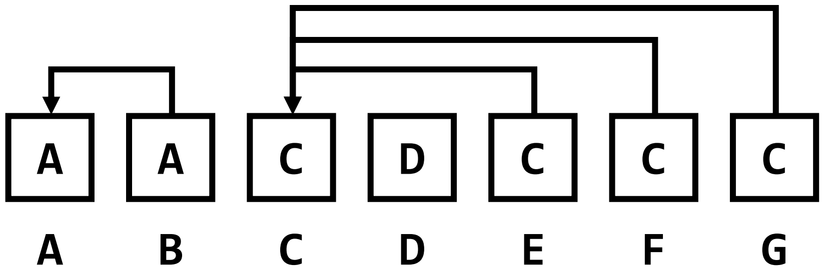
Disjoint sets
{ A B }
{ C F }
{ D }
{ E G }

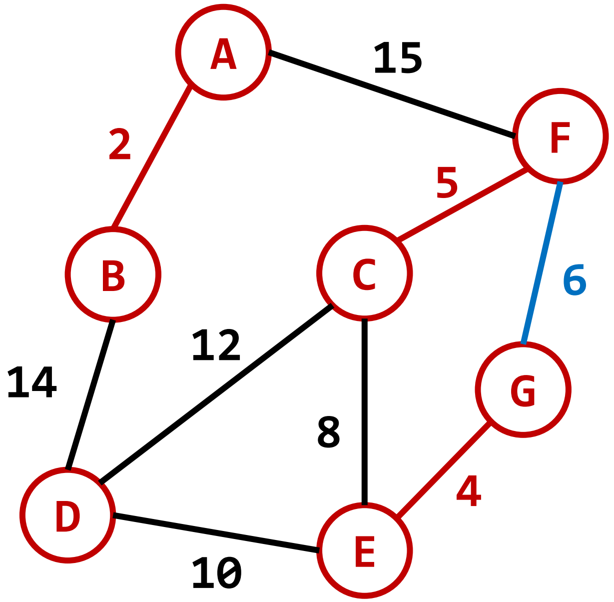
Kruskal's Algorithm

- The key idea is to add edge one by one into the edge set
  1. Start from a zero-edge subgraph T=(V,F) where V = { all vertices } and F = Ø
  2. Pick the edge of the minimum **addition cost** not form a cycle in T
  3. **Add the edge** into T
  4. Repeat 2 & 3 steps

If **the edge** connects two disjoint sets, merge them
Otherwise, skip the edge

X → **Y** means
X belongs to the set including **Y**





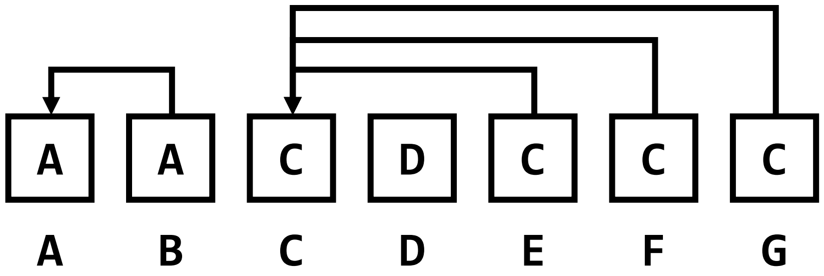
Disjoint sets
{ A B }
{ C F }
{ D }
{ E G }

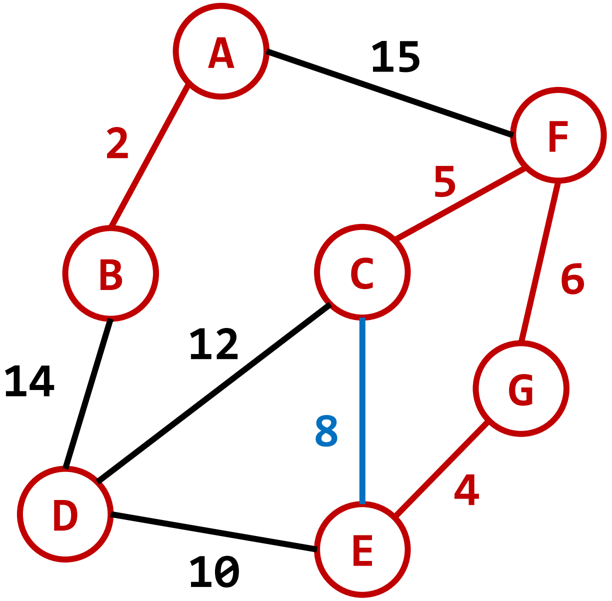
Kruskal's Algorithm

- The key idea is to add edge one by one into the edge set
  1. Start from a zero-edge subgraph `T=(V,F)` where `V` = { all vertices } and `F` = Ø
  2. Pick the edge of the minimum **addition cost** not form a cycle in `T`
  3. **Add the edge** into `T`
  4. Repeat 2 & 3 steps

If **the edge** connects two disjoint sets, merge them
Otherwise, skip the edge

X → **Y** means
X belongs to the set including **Y**





Disjoint sets

{ A B }

{ C E F G }

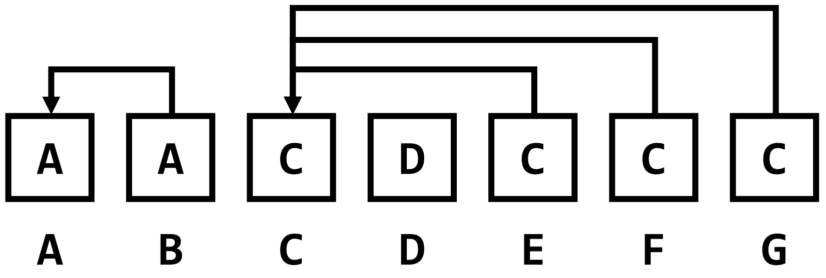
{ D }

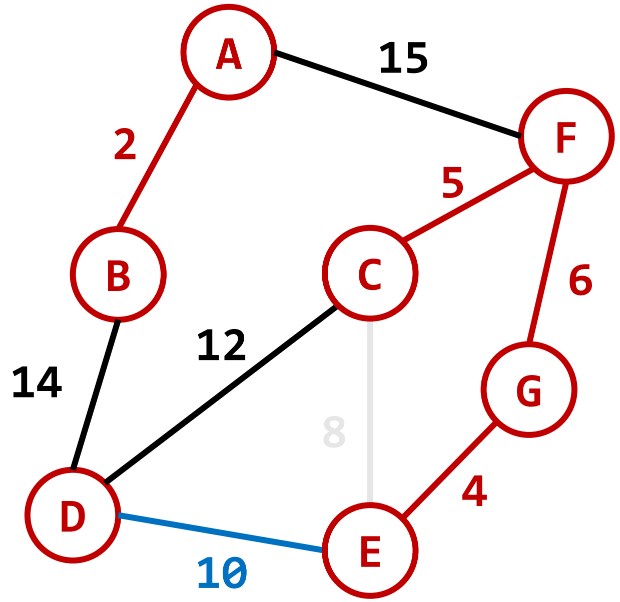
Kruskal's Algorithm

- The key idea is to add edge one by one into the edge set
  1. Start from a zero-edge subgraph `T=(V,F)` where V = { all vertices } and F = ∅
  2. Pick the edge of the minimum **addition cost** not form a cycle in T
  3. **Add the edge** into T
  4. Repeat 2 & 3 steps

If **the edge** connects two disjoint sets, merge them
Otherwise, skip the edge

X → **Y** means
X belongs to the set including **Y**





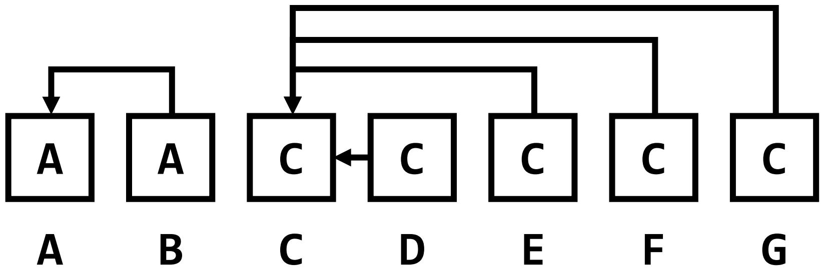
Disjoint sets
{ A B }
{ C E F G }
{ D }

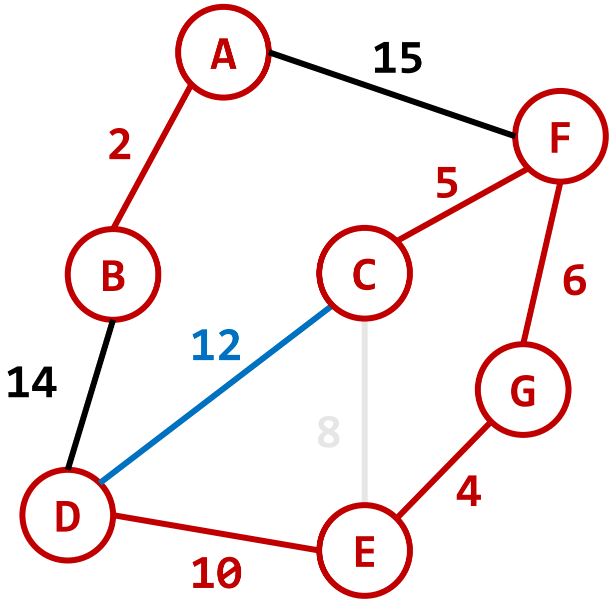
Kruskal's Algorithm

- The key idea is to add edge one by one into the edge set
  1. Start from a zero-edge subgraph T=(V,F) where V = { all vertices } and F = Ø
  2. Pick the edge of the minimum **addition cost** not form a cycle in T
  3. **Add the edge** into T
  4. Repeat 2 & 3 steps

If **the edge** connects two disjoint sets, merge them
Otherwise, skip the edge

X → **Y** means
X belongs to the set including **Y**





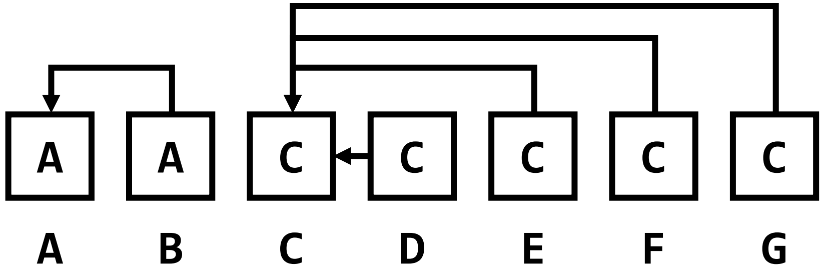
Disjoint sets
{ A B }
{ C D E F G }

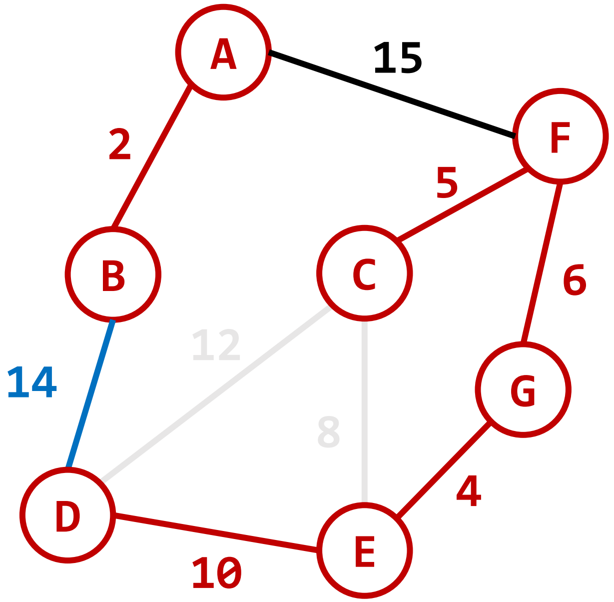
Kruskal's Algorithm

- The key idea is to add edge one by one into the edge set
  1. Start from a zero-edge subgraph `T=(V,F)` where `V` = { all vertices } and `F` = ∅
  2. Pick the edge of the minimum **addition cost** not form a cycle in `T`
  3. **Add the edge** into `T`
  4. Repeat 2 & 3 steps

If **the edge** connects two disjoint sets, merge them
Otherwise, skip the edge

X → **Y** means
X belongs to the set including **Y**





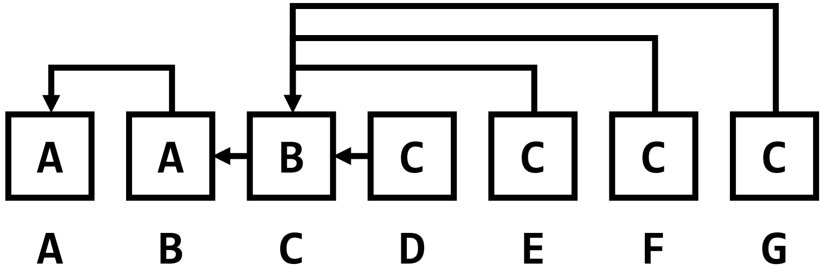
Disjoint sets
{ A B }
{ C D E F G }

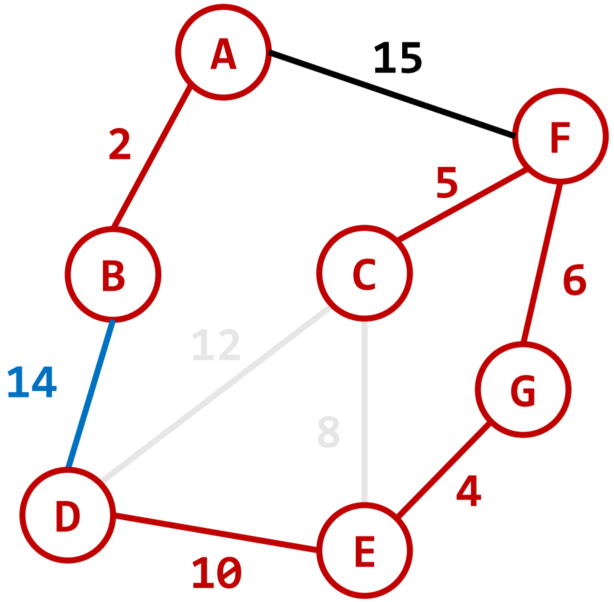
Kruskal's Algorithm

- The key idea is to add edge one by one into the edge set
  1. Start from a zero-edge subgraph T=(V,F) where V = { all vertices } and F = Ø
  2. Pick the edge of the minimum **addition cost** not form a cycle in T
  3. **Add the edge** into T
  4. Repeat 2 & 3 steps

If **the edge** connects two disjoint sets, merge them
Otherwise, skip the edge

X → **Y** means
X belongs to the set including **Y**





Disjoint sets

{ A B }

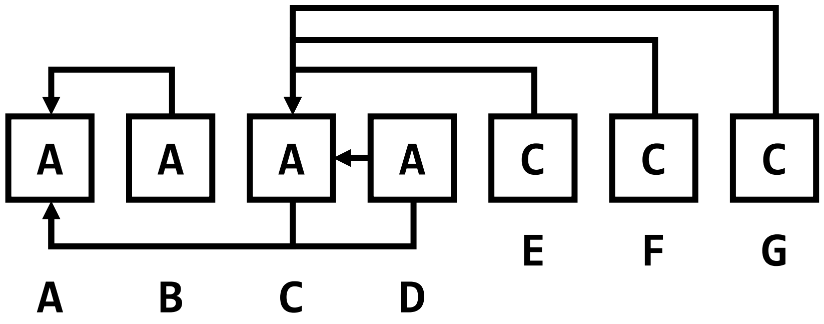
{ C D E F G }

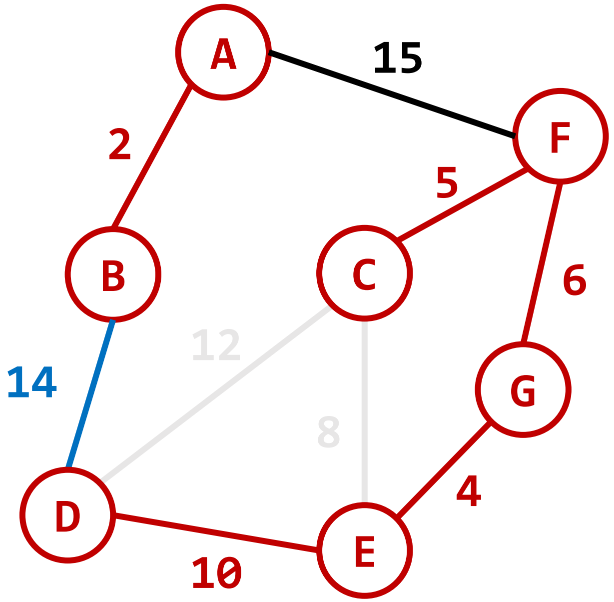
Kruskal's Algorithm

- The key idea is to add edge one by one into the edge set
  1. Start from a zero-edge subgraph T=(V,F) where V = { all vertices } and F = Ø
  2. Pick the edge of the minimum **addition cost** not form a cycle in T
  3. **Add the edge** into T
  4. Repeat 2 & 3 steps

If **the edge** connects two disjoint sets, merge them
Otherwise, skip the edge

X → **Y** means
X belongs to the set including **Y**





Disjoint sets
{ A B }
{ C D E F G }

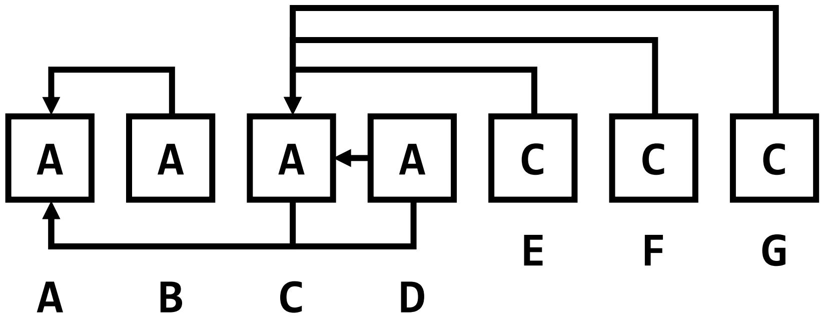
Kruskal's Algorithm

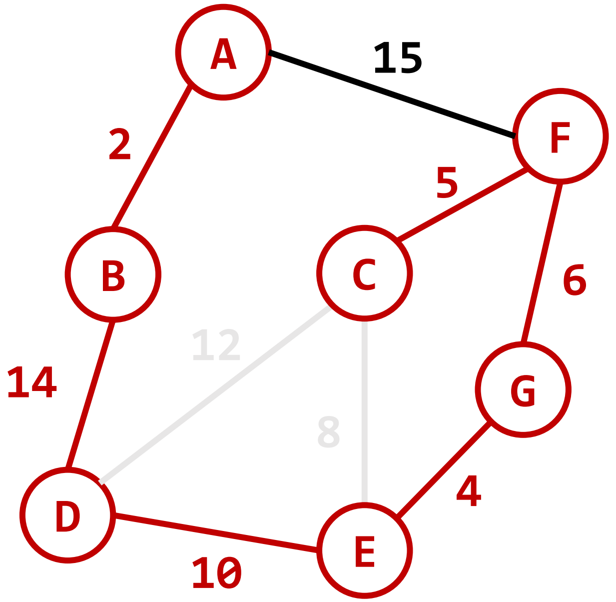
- The key idea is to add edge one by one into the edge set
  1. Start from a zero-edge subgraph T=(V,F) where V = { all vertices } and F = Ø
  2. Pick the edge of the minimum **addition cost** not form a cycle in T
  3. **Add the edge** into T
  4. Repeat 2 & 3 steps

If **the edge** connects two disjoint sets, merge them
Otherwise, skip the edge

Disjoint sets
{ A B C D E F G }

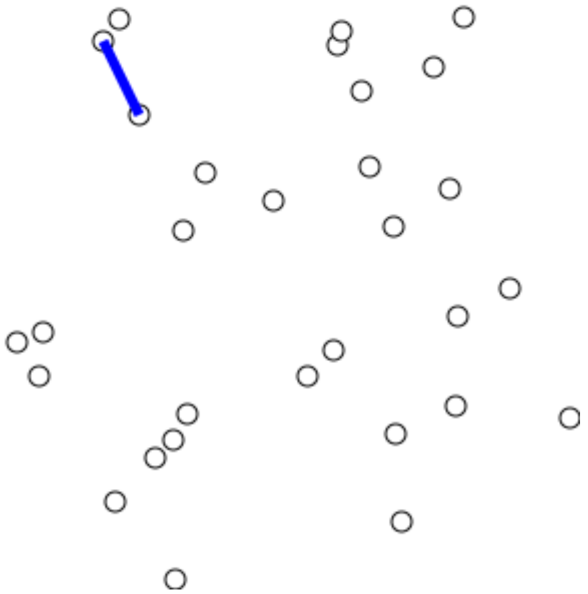
X → **Y** means
X belongs to the set including **Y**



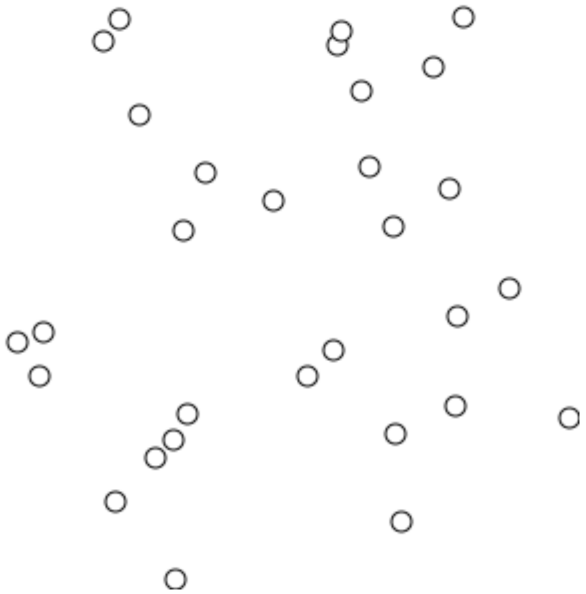


Prim's vs Kruskal's Algorithms

Prim's Algorithm



Kruskal's Algorithm



- Prim's algorithm is based on **vertex addition** and uses **segment tree** structure
- Kruskal's algorithm is based on **edge addition** and uses **disjoint-set** structure
- Both algorithms has $O(|E|\log|V|)$ time complexity

Any Questions?