# ALGORITHMS AND DATA STRUCTURES II

<u>Lecture 4</u>

**Spanning Tree,**
**Weighted Graphs,**
**Prim's and Kruskal's algorithms.**



**Lecturer: K. Markov**

markov@u-aizu.ac.jp

# SPANNING TREE

- Assume you have an **undirected graph**

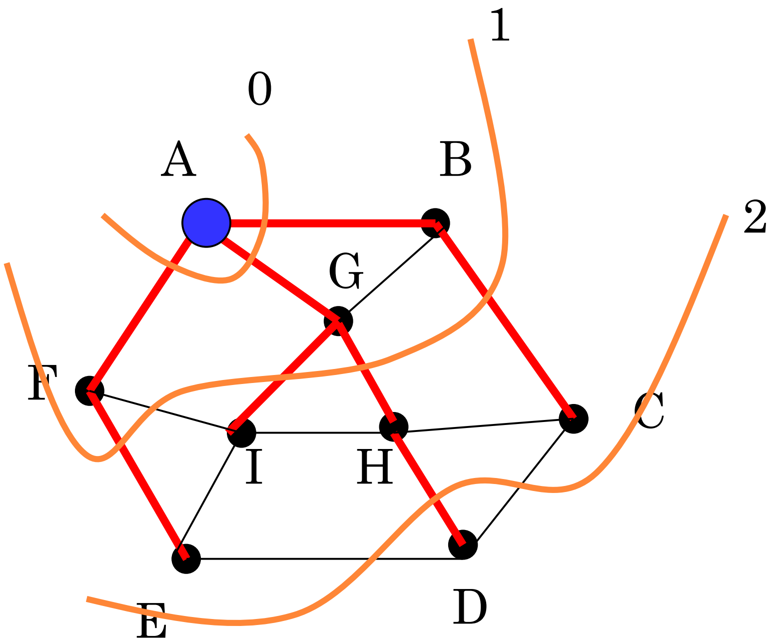
  $$\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E})$$

- **Spanning tree** of graph $\boldsymbol{G}$ is the tree

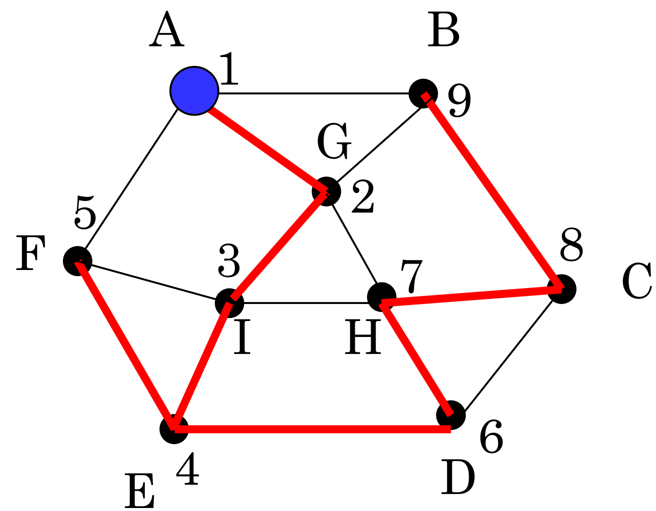
  $$\boldsymbol{T} = (\boldsymbol{V}, \boldsymbol{E_T} \subseteq \boldsymbol{E}, \boldsymbol{R})$$

  - Tree has the same set of nodes.
  - All tree edges are graph edges.
  - Root of the tree is $\boldsymbol{R}$.
- **Think:** "smallest set of edges needed to connect everything together".

# SPANNING TREE



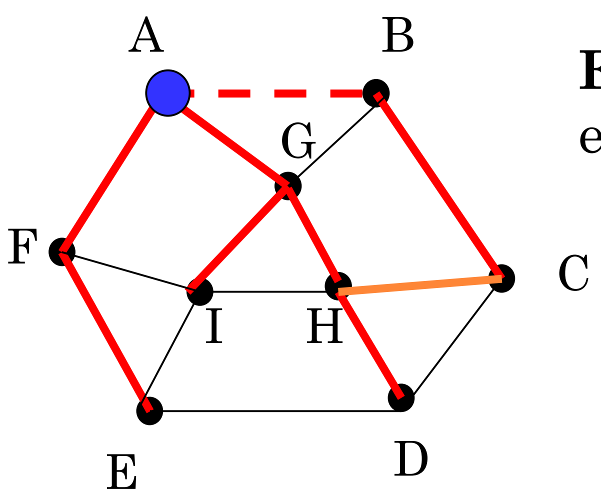

Breadth-first Spanning Tree




Depth-first spanning tree

# SPANNING TREE

- **Properties:**
  - In any tree $\boldsymbol{T} = (\boldsymbol{V}, \boldsymbol{E})$, $|\boldsymbol{E}| = |\boldsymbol{V}| - \mathbf{1}$.
  - For any edge $\boldsymbol{e}$ in $\boldsymbol{G}$ but not in $\boldsymbol{T}$, there is a simple cycle $\boldsymbol{Y}$ containing only edge $\boldsymbol{e}$ and edges in spanning tree.
  - Moreover, inserting edge $\boldsymbol{e}$ into $\boldsymbol{T}$ and deleting any edge in $\boldsymbol{Y}$ gives another spanning tree $\boldsymbol{T'}$.




**EXAMPLE:**

edge $(H, C)$:

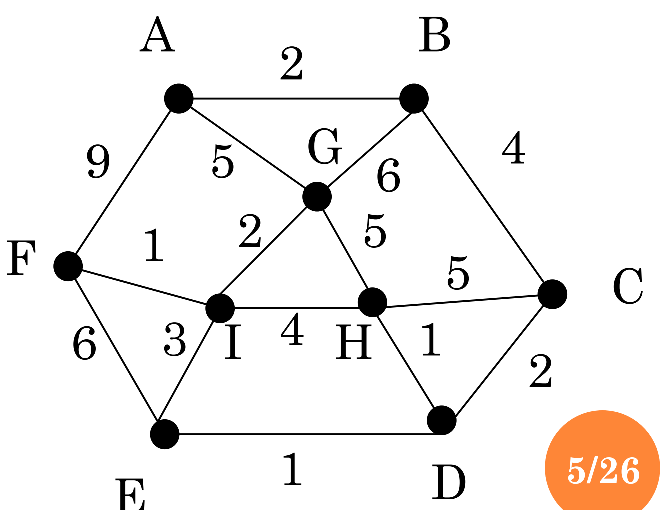
simple cycle is $(H, C, B, A, G, H)$

adding $(H, C)$ to $\boldsymbol{T}$ and deleting $(A, B)$

gives another spanning tree

# WEIGHTED GRAPHS

- **Definition:**
  - A **weighted graph** is a graph $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E})$ with real valued weights assigned to each edge.
  - Equivalently, a weighted graph is a triple $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E}, \boldsymbol{W})$, where $\boldsymbol{V}$ is the set of vertices, $\boldsymbol{E}$ is the set of edges, and $\boldsymbol{W}$ is the set of weights. The weights on edges are also called **distances** or **costs**.




Weighted Graph

# WEIGHTED GRAPHS

- **Representation:**
  - A weighted graph $\boldsymbol{G}(\boldsymbol{V},\boldsymbol{E},\boldsymbol{W})$ can be represented by a **distance matrix**

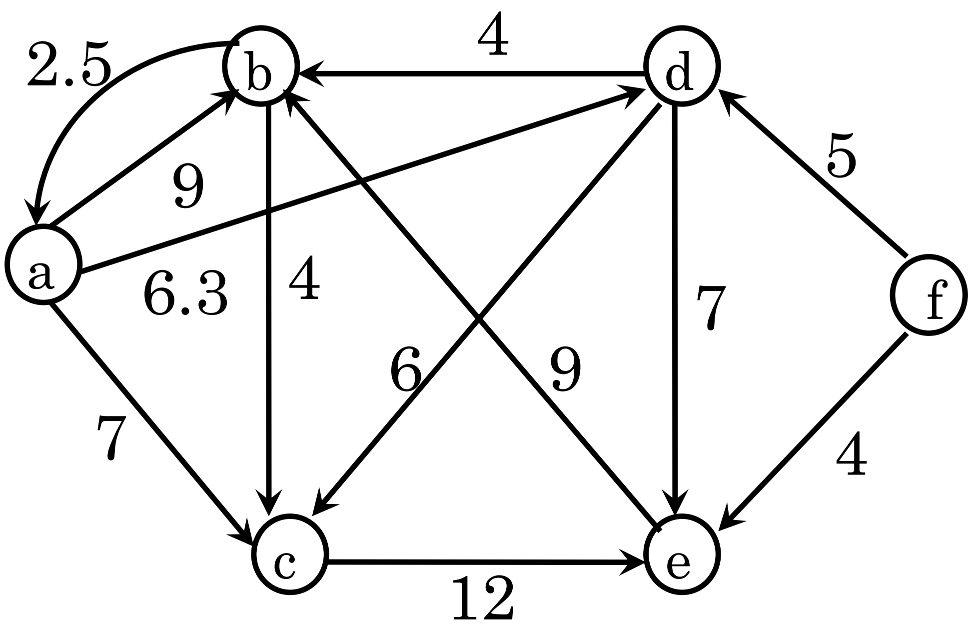
$$D_{n\times n} = \begin{bmatrix} d(1,1) & \dots & d(1,n) \\ \dots & \dots & \dots \\ d(n,1) & \dots & d(n,n) \end{bmatrix} \quad n=|V|$$

where $d(i,i) = 0$,

and for $1 \le i \ne j \le n$, if edge $(i,j) \in E$, then $d(i,j)$ is the weight of $(i,j)$, otherwise $d(i,j)$ is infinite $\infty$ (a sufficiently large number in practice).

# WEIGHTED GRAPHS

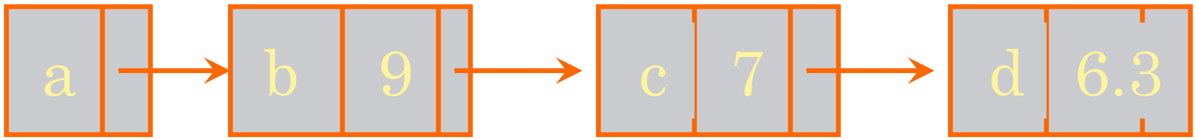
- **Representation:**




Distance Matrix

| | a | b | c | d | e | f |
|---|---|---|---|---|---|---|
| **a** | 0 | 9 | 7 | 6.3 | ∞ | ∞ |
| **b** | 2.5 | 0 | 4 | ∞ | ∞ | ∞ |
| **c** | ∞ | ∞ | 0 | ∞ | 12 | ∞ |
| **d** | ∞ | 4 | 6 | 0 | 7 | ∞ |
| **e** | ∞ | 9 | ∞ | ∞ | 0 | ∞ |
| **f** | ∞ | ∞ | ∞ | 5 | 4 | 0 |

Adjacency list



# MINIMUM SPANNING TREE (MST)

- Let $\boldsymbol{T}(\boldsymbol{V}', \boldsymbol{E}')$ be a spanning tree of a weighted graph $\boldsymbol{G}$ and

$$\boldsymbol{W}(\boldsymbol{T}) = \sum_{(\boldsymbol{v},\boldsymbol{w})\in \boldsymbol{E}'} \boldsymbol{W}(\boldsymbol{v},\boldsymbol{w})$$

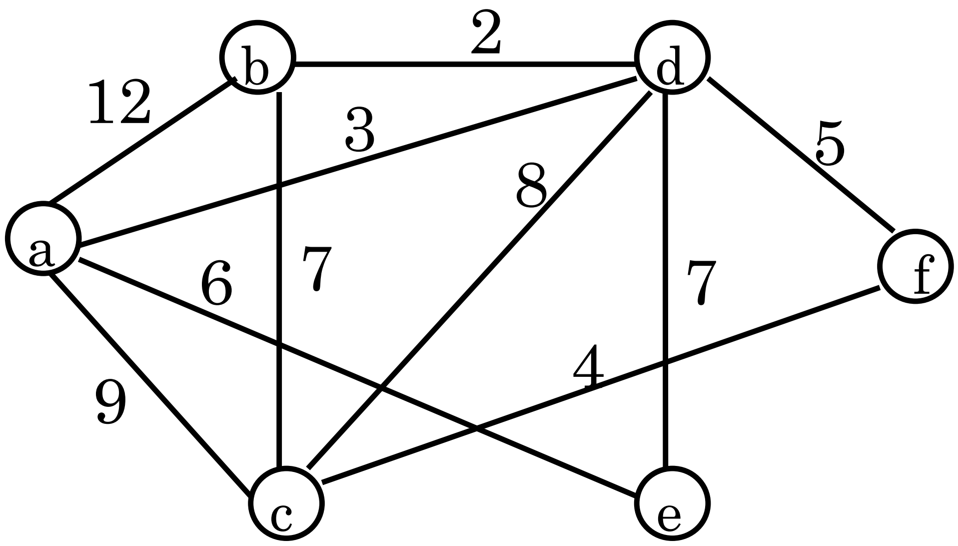
  be the sum of weights of edges in $\boldsymbol{T}$, where $\boldsymbol{W}(\boldsymbol{v},\boldsymbol{w})$ denotes the weight of edge $(\boldsymbol{v},\boldsymbol{w})$**.**

- A **minimum spanning tree** of $\boldsymbol{G}$ is a spanning tree $\boldsymbol{T}^{\boldsymbol{m}}$ of $\boldsymbol{G}$ such that

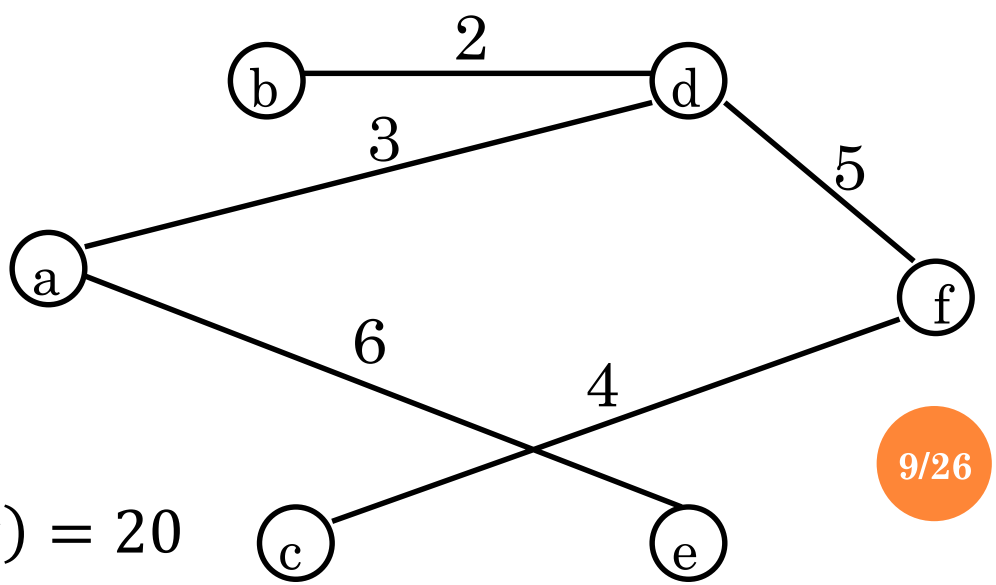
$$\boldsymbol{W}(\boldsymbol{T}^{\boldsymbol{m}}) = \mathbf{min}_{\boldsymbol{T}}\{\boldsymbol{W}(\boldsymbol{T})\}$$

# MINIMUM SPANNING TREE (MST)

**Weighted Graph G**




**Minimum Spanning Tree of G**

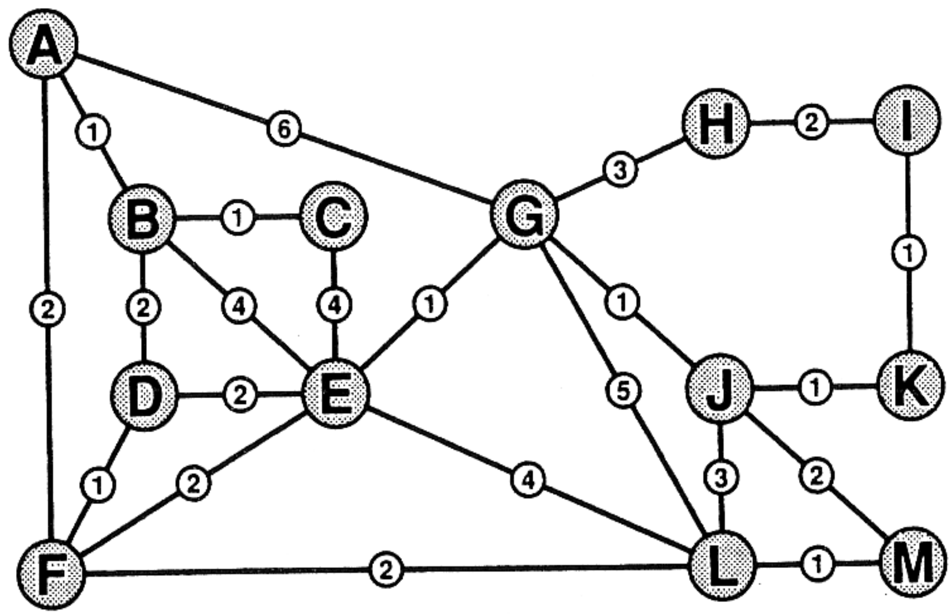



$$W(T^m) = \sum_{(v,w)\in E'} W(v,w) = 20$$

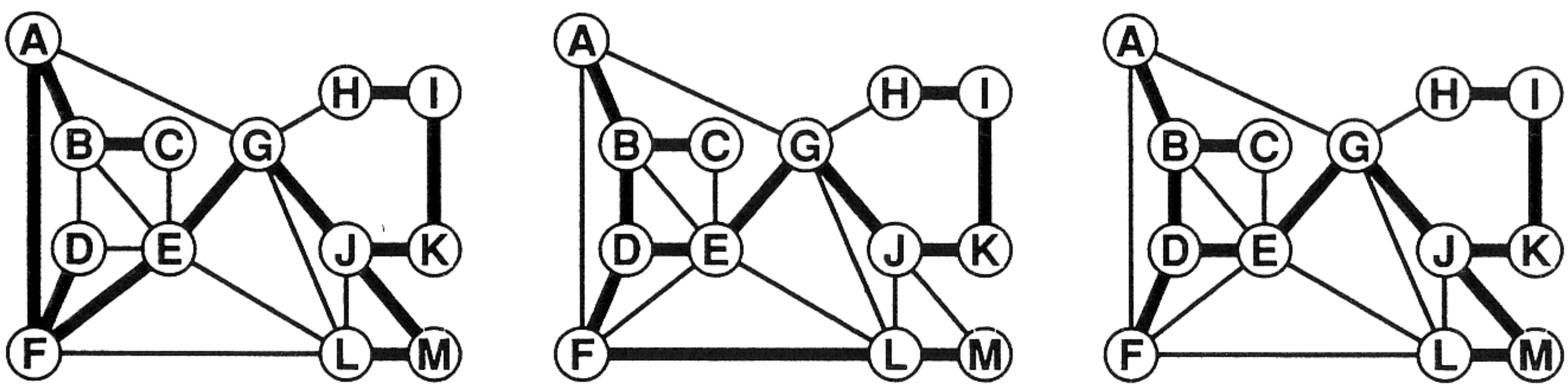
# MINIMUM SPANNING TREE (MST)

- **Minimum spanning tree** is useful when we attempt to minimize the cost of connecting all the nodes.
- **Applications:**
  - Constructing electric power networks or telephone networks.
  - Making printed circuit boards (PCBs).
  - Etc.
- **Note:** Minimum spanning tree need not to be unique. (simple examples)

# MINIMUM SPANNING TREE (MST)




**Weighted Graph G**




**Multiple Minimum Spanning Trees of G**

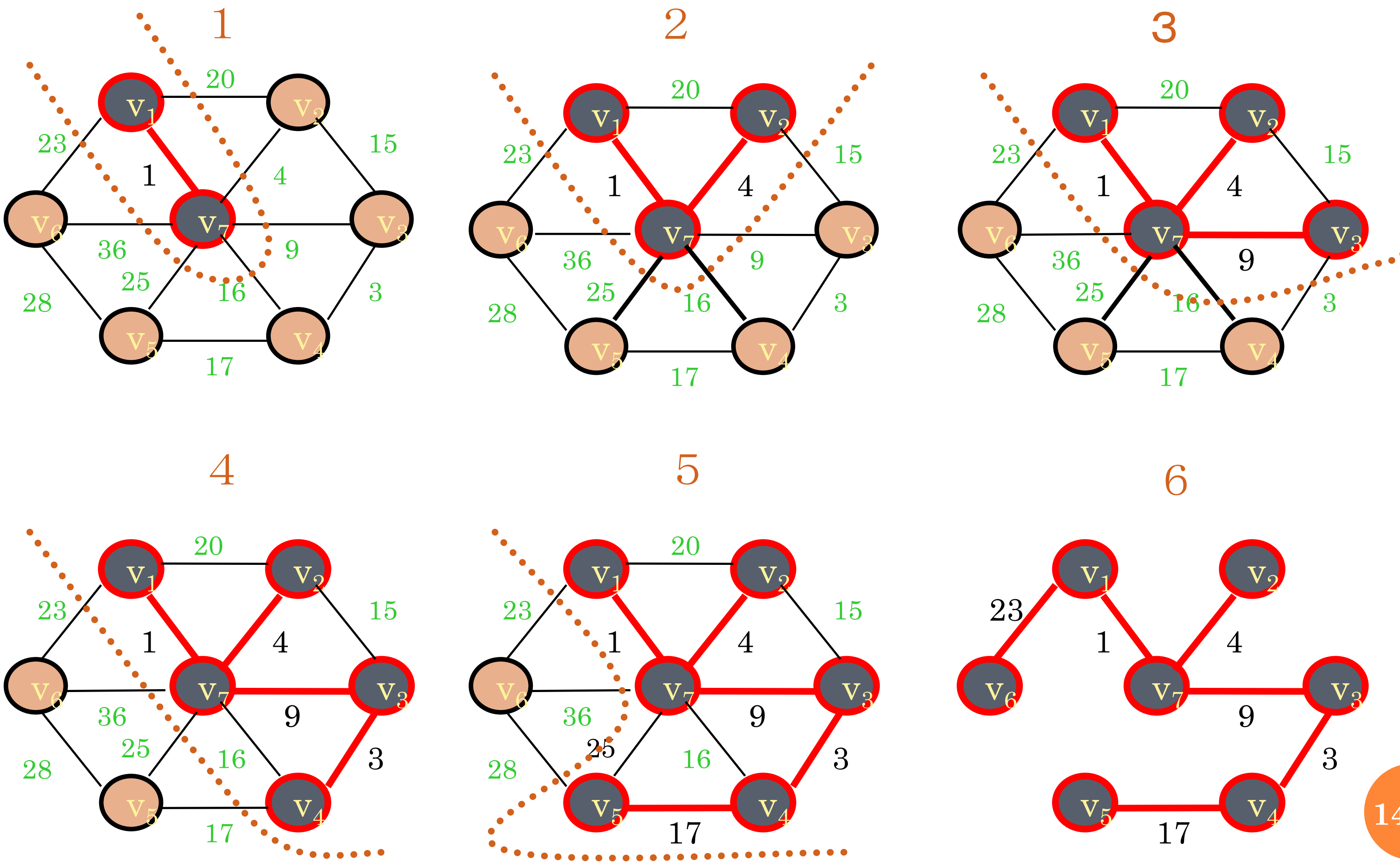
# MINIMUM SPANNING TREE (MST)

- **Building MST** – two strategies:
  - **Prim's algorithm** – start with a root node ***s*** and try to grow a tree from ***s*** outward. At each step, add the node that can be attached as cheaply as possible to the partial tree we already have.
  - **Kruskal's algorithm** – start with no edges and successively insert edges from ***E*** in order of increasing cost. If an edge makes cycle when added, skip this edge.

# PRIM'S ALGORITHM

1) Pick an arbitrary vertex $\boldsymbol{r}$ of $\boldsymbol{G}(\boldsymbol{V},\boldsymbol{E})$ as the root of the minimum spanning tree of $\boldsymbol{G}$. Assume a partial solution (spanning tree) $\boldsymbol{T}$ has been obtained (initially, $\boldsymbol{T} = \{\boldsymbol{r}\}$).
2) Choose an edge $(\boldsymbol{v},\boldsymbol{w})$ such that $\boldsymbol{v} \in \boldsymbol{T}$, $\boldsymbol{w} \in \boldsymbol{V}-\boldsymbol{T}$, and the weight of edge $(\boldsymbol{v},\boldsymbol{w})$ is the minimum among that of edges from the nodes of $\boldsymbol{T}$ to nodes of $\boldsymbol{V}-\boldsymbol{T}$.
3) Add the node $\boldsymbol{w}$ into $\boldsymbol{T}$.
4) Repeat the above 2) and 3) until $\boldsymbol{T} = \boldsymbol{V}$.

# PRIM'S ALGORITHM



# PRIM'S ALGORITHM

- If the graph is represented by an **adjacency (distance) matrix**, the time complexity of Prim's algorithm is $\boldsymbol{O(V^2)}$.

- Prim's algorithm can be made more efficient by maintaining the graph using **adjacency lists** and keeping a **priority queue** of the nodes not in $\boldsymbol{T}$. Under this implementation, the time complexity of Prim's algorithm is $\boldsymbol{O(V \log V + E \log V) = O(E \log V)}$.

# PRIM'S ALGORITHM

- Implementation:

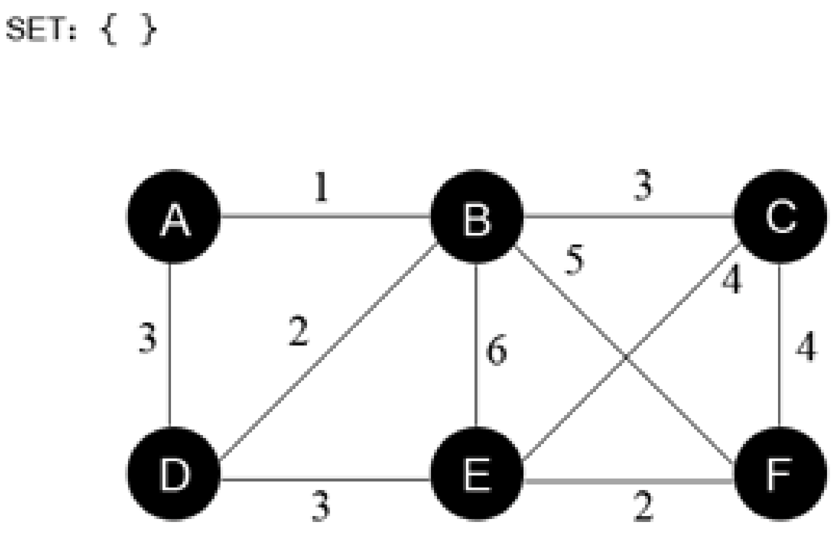
```
def MST-PRIM (G, w, r)
//  Graph G with set of nodes G.V, weight matrix w and
// root node r. MST is the edges set A={(v,v.π), v ∈ V-r}.
   for each u ∈ G.V:
      u.key = ∞
      u.π = NIL
   r.key = 0
   Q = Min-Priority-Queue (G.V)
   while Q ≠ ∅:
      u = Extract-Min (Q)
      for each v ∈ G.Adj[u]:
         if v ∈ Q and w(u,v) < v.key:
            v.π = u
            v.key = w(u,v)
```

# PRIM'S ALGORITHM

- Implementation notes:
  - During execution of the algorithm, all nodes that are **NOT** in the **MST**, reside in the **minimum priority queue** based on the *key* attribute.
  - For each node $\boldsymbol{v}$, the attribute $\boldsymbol{v.key}$ is the minimum weight of any edge connecting $\boldsymbol{v}$ to a node in the **MST**.
  - If there is no edge $\boldsymbol{v.key} = \infty$.
  - The attribute $\boldsymbol{v.\pi}$ names the parent of $\boldsymbol{v}$ in the **MST**.

# PRIM'S ALGORITHM

- Animated example:



# KRUSKAL'S ALGORITHM

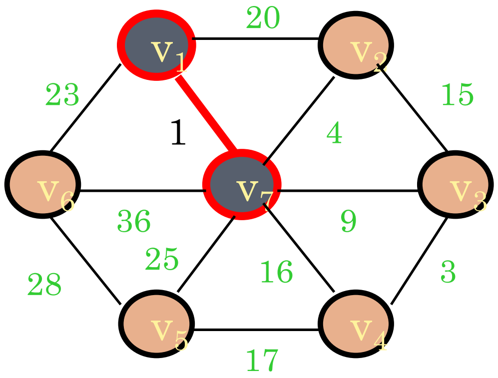
1) Pick the cheapest edge available and add it to the **MST**

$$\boldsymbol{e_0} = \min_{(\boldsymbol{v},\boldsymbol{u})} \boldsymbol{w}(\boldsymbol{v},\boldsymbol{u}), \ \boldsymbol{A} = \{\boldsymbol{e_0}\}$$

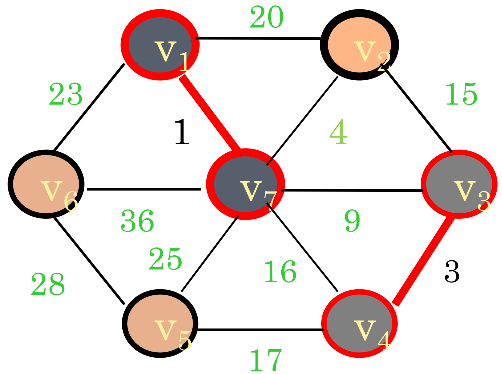
2) Choose next cheapest edge $\boldsymbol{e} = (\boldsymbol{v},\boldsymbol{w})$
3) If adding $\boldsymbol{e}$ to the $\boldsymbol{A}$ makes a cycle, do not add it.
4) Repeat the above 2) and 3) until all edges are chosen.

# KRUSKAL'S ALGORITHM

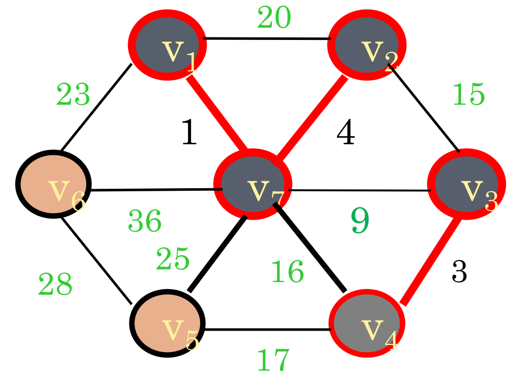
1



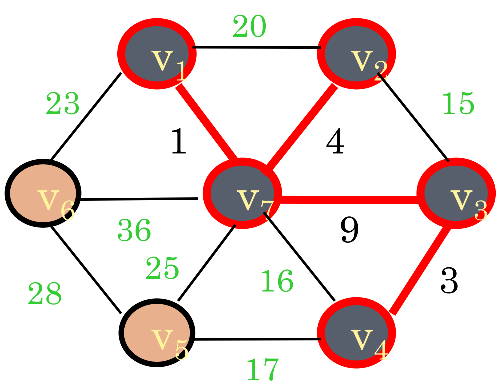

2



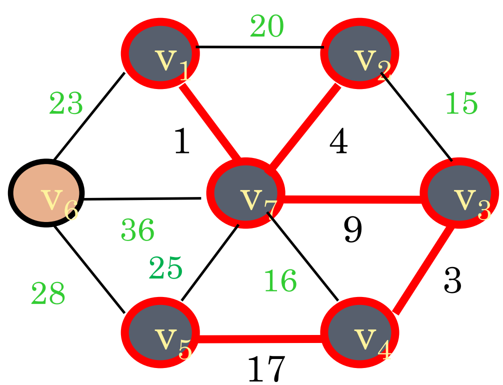

3



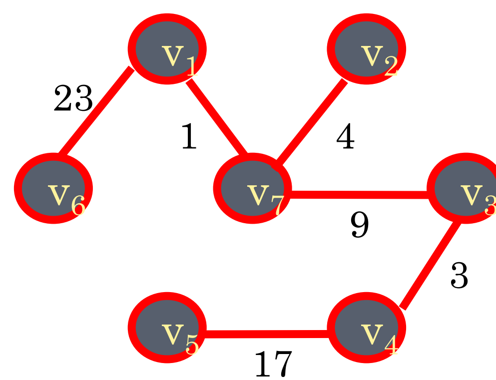

4




5




6



# KRUSKAL'S ALGORITHM

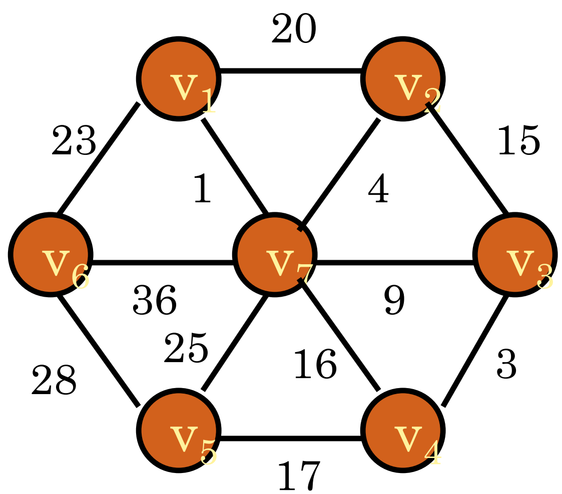
- Implementation:

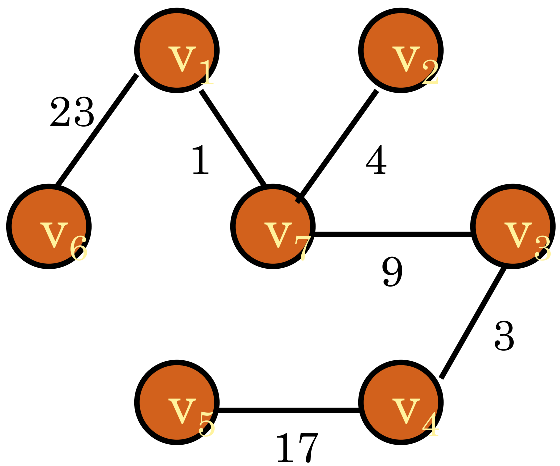
```
def  MST-KRUSKAL (G, w)
//  Graph G with set of nodes G.V, weight matrix w.
//  MST is the edges set A={}.
    A = ∅
    for each v ∈ G.V:
        MAKE-SET (v)
    Sort edges of G.E into non-decreasing order by weight w
    for each edge (u, v) ∈ G.E, taken in non-decreasing order of w:
        if FIND-SET (u) ≠ FIND-SET (v):
            A = A ∪ {(u, v)}
            UNION (u, v)
    return A
```

# KRUSKAL'S ALGORITHM

- Implementation notes.
  - **UNION-FIND** data structure:
    - Given a node ***u*** the operation **FIND-SET (u)** will return the name of the set containing ***u***.
    - To test if two nodes ***u*** and ***v*** are in the same set, we simply check if **FIND-SET(u) = FIND-SET(v)**
    - The operation **UNION (u, v)** will take two sets containing ***u*** and ***v*** respectively and will merge them into a single set.
    - To make a set from one or several nodes, we use the **MAKE-SET** () operation.

# KRUSKAL'S ALGORITHM

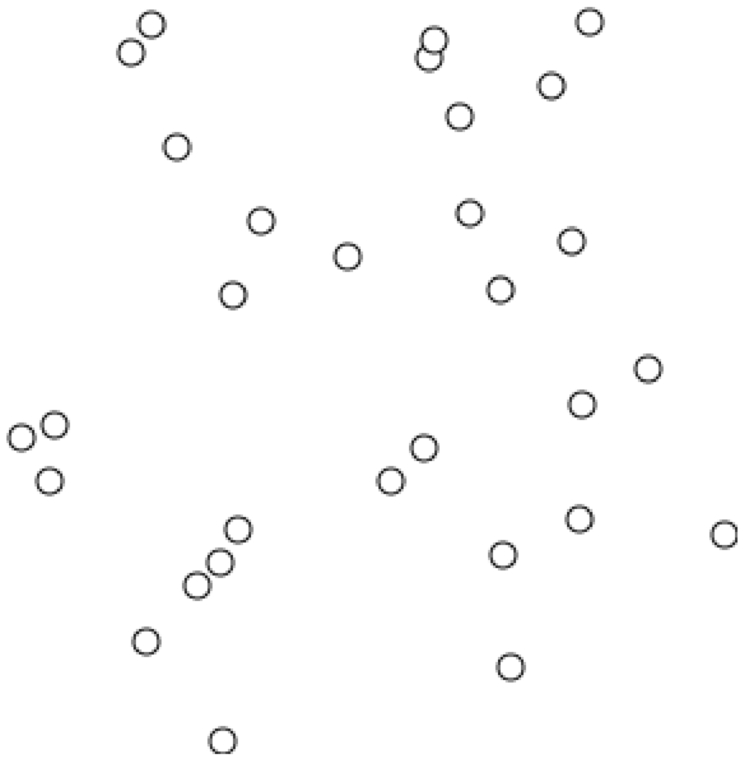



| Edge | Action | Sets |
|---|---|---|
| | | $\{v_1\},\{v_2\},\{v_3\},\{v_4\},\{v_5\},\{v_6\},\{v_7\}$ |
| $(v_1, v_7)$ | Add | $\{v_1,v_7\},\{v_2\},\{v_3\},\{v_4\},\{v_5\},\{v_6\}$ |
| $(v_3, v_4)$ | Add | $\{v_1,v_7\},\{v_2\},\{v_3,v_4\},\{v_5\},\{v_6\}$ |
| $(v_2, v_7)$ | Add | $\{v_1,v_2,v_7\},\{v_3,v_4\},\{v_5\},\{v_6\}$ |
| $(v_3, v_7)$ | Add | $\{v_1,v_2,v_3,v_4,v_7\},\{v_5\},\{v_6\}$ |
| $(v_2, v_3)$ | Reject | |
| $(v_4, v_7)$ | Reject | |
| $(v_4, v_5)$ | Add | $\{v_1,v_2,v_3,v_4,v_5,v_7\},\{v_6\}$ |
| $(v_1, v_2)$ | Reject | |
| $(v_1, v_6)$ | Add | $\{v_1,v_2,v_3,v_4,v_5,v_6,v_7\}$ |

# KRUSKAL'S ALGORITHM

- Animated example based on Euclidean distance:



# KRUSKAL'S ALGORITHM

- Complexity.
  - Initializing set $\boldsymbol{A}$ takes $\boldsymbol{O}(1)$.
  - Making $|\boldsymbol{V}|$ sets takes $\boldsymbol{O}(\boldsymbol{V})$ time.
  - Time to sort the edges by weight is $\boldsymbol{O}(\boldsymbol{E} \log \boldsymbol{E})$.
  - There are $|\boldsymbol{E}|$ FIND-SET and UNION operations taking $\boldsymbol{O}(\boldsymbol{E})$ time.
  - Since the graph is connected, $|\boldsymbol{E}| \geq |\boldsymbol{V}| - 1$ and $|\boldsymbol{E}| < |\boldsymbol{V}|^2$, $\boldsymbol{\log |V|^2 = 2 \log |V|}$ which is $\boldsymbol{O(\log V)}$.
  - Total running time is $\boldsymbol{O(E \log V)}$.

# That's All for Today!