ALGORITHMS AND DATA STRUCTURES II

Lecture 4

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

Lecturer: K. Markov

markov@u-aizu.ac.jp

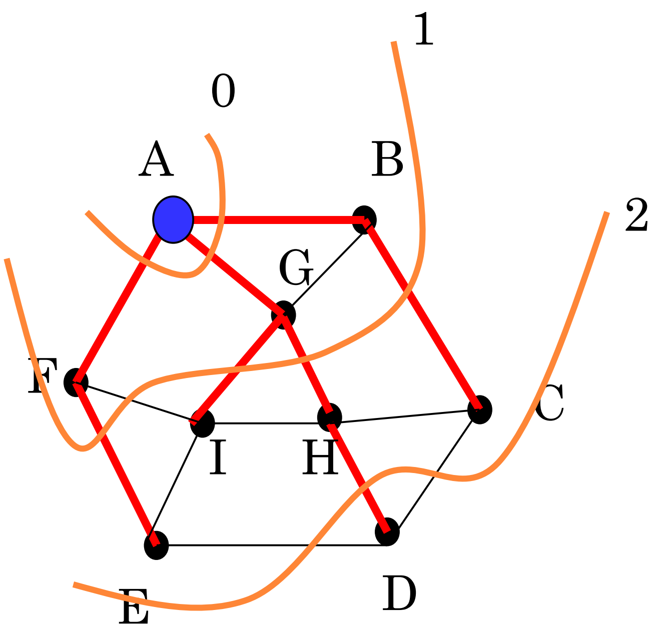
SPANNING TREE

- Assume you have an **undirected graph** $\boldsymbol{G = (V, E)}$
- **Spanning tree** of graph $\boldsymbol{G}$ is a subgraph $\boldsymbol{T}$ of $\boldsymbol{G}$

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

  - It has the same set of nodes.
  - All its edges are graph edges.
  - It is a **tree** - not cycles, $\boldsymbol{|E_T| = |V| - 1}$
  - **Root** of the tree is $\boldsymbol{R \subseteq V}$.
- **Think:** “smallest set of edges needed to connect everything together”.

SPANNING TREE WITH BFS



Breadth-first Spanning Tree

Steps to create a Spanning Tree

• Start with arbitrarily chosen vertex of the graph as the root.
• Add all edges incident to the vertex along with the other vertex connected to each edge
– The new vertices added become the vertices at level 1 in the spanning tree.

• For each vertex at level 1 add each edge incident to this vertex and the other vertex connected to the edge to the tree
– The new vertices added become the new vertices at level 2 in the spanning tree.

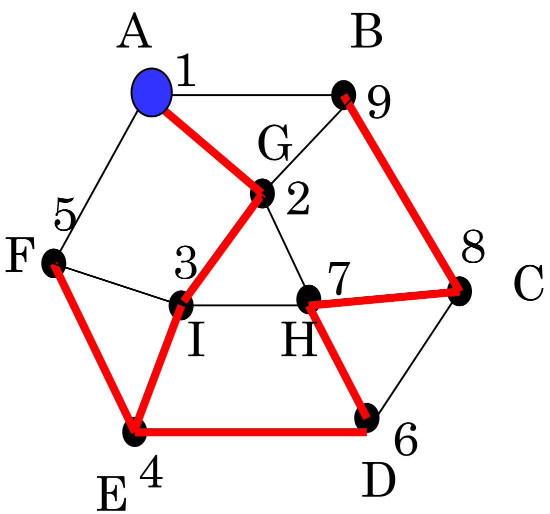
• Repeat the procedure until all vertices in the graph have been added.

SPANNING TREE WITH BFS

Algorithm BFS (G : connected graph with vertices $v_1, v_2, \ldots, v_n$)

T = tree consisting only of vertex v_1
L = empty list
put v_1 in the list L of unprocessed vertices
while L is not empty **do**
  remove the first vertex, v, from L
  for each neighbor w of v **do**
    if w is not in L and not in T **then**
      add w to the end of the list L
      add w and edge (v, w) to T
    end if
  end for
end while

SPANNING TREE WITH DFS



Depth-first spanning tree

Steps to create Spanning Tree

• Start with arbitrarily chosen vertex of the graph as the root.

• Form a path from the root by successively adding vertices and edges where

– Each new edge is incident with the last vertex in the path.

– Vertices are not already in the path.

– Continue adding vertices and edges to this path as long as possible,

• If all vertices are included in the path, then “Done”.

• Otherwise, move back to the next to last vertex in the path and, if possible, form a new path starting at this vertex passing through vertices that were not already visited. If this cannot be done, move back another vertex and try again.

• Repeat this procedure until no more edges can be added.

SPANNING TREE WITH DFS

Algorithm DFS (G : connected graph with vertices $v_1, v_2, \ldots, v_n$)

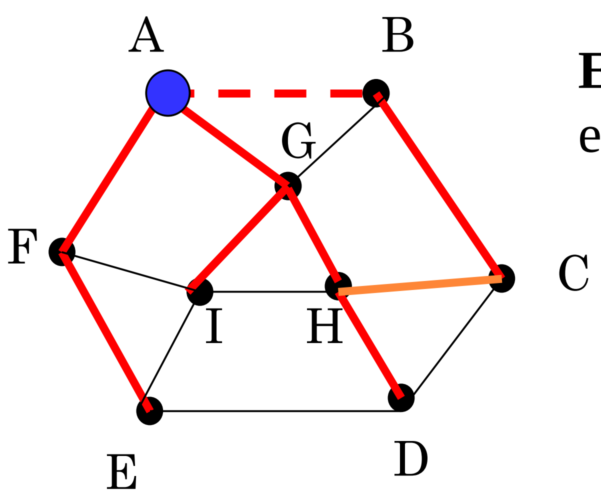
1: T $=$ tree consisting only of the vertex v_1
2: visit(v_1)

Procedure visit(v : vertex of G)

1: **for** each vertex w adjacent to v and not yet in T **do**
2: add vertex w and edge (v, w) to T
3: visit(w)
4: **end for**

SPANNING TREE

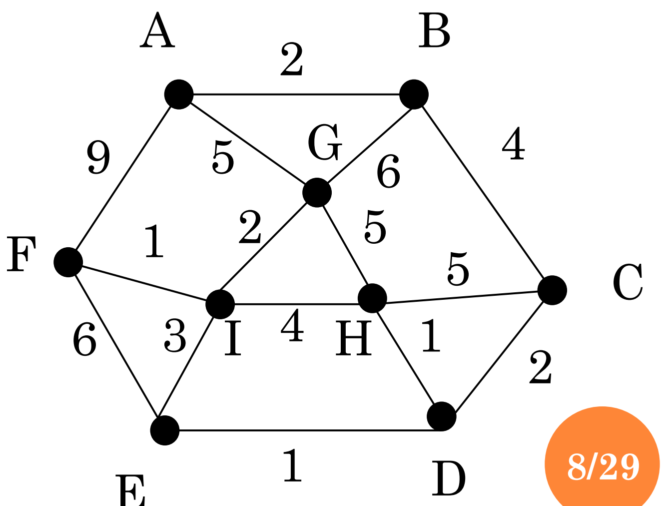
- **Properties:**
  - In any tree $\boldsymbol{T} = (\boldsymbol{V}, \boldsymbol{E})$, $|\boldsymbol{E}| = |\boldsymbol{V}| - \mathbf{1}$.
  - For any edge $\boldsymbol{e}$ in $\boldsymbol{G}$ but not in $\boldsymbol{T}$, there is a simple cycle $\boldsymbol{Y}$ containing only edge $\boldsymbol{e}$ 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**

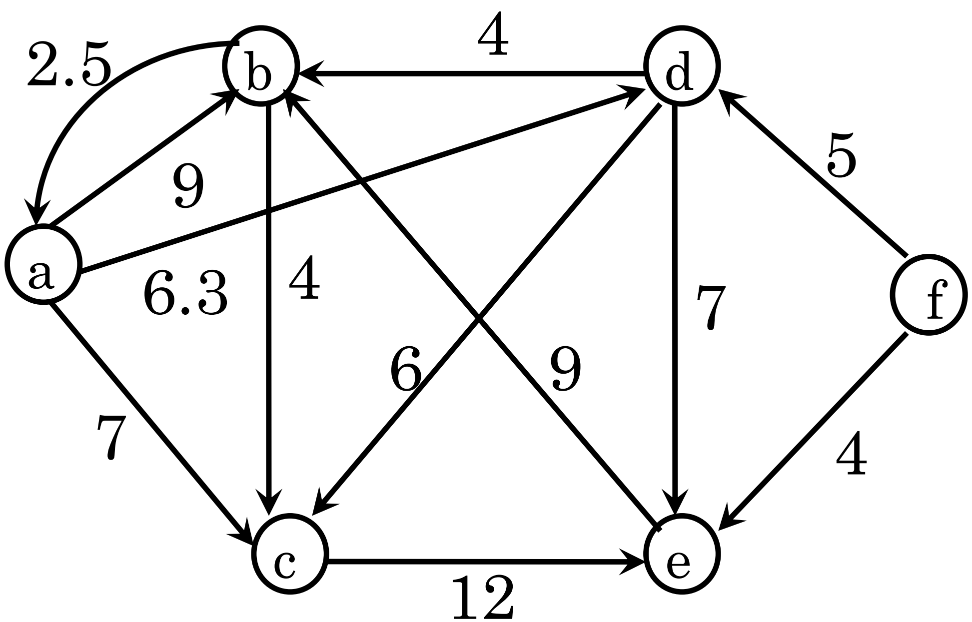
$$D_{n\times n} = \begin{bmatrix} d(1,1) & \dots & d(1,n) \\ \dots & \dots & \dots \\ d(n,1) & \dots & d(n,n) \end{bmatrix} \quad \text{n=|V|}$$

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

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

WEIGHTED GRAPHS

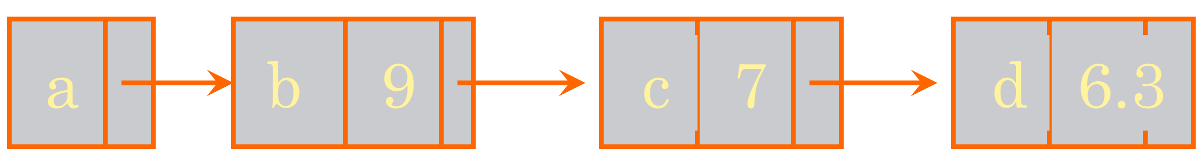
- **Representation:**



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

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

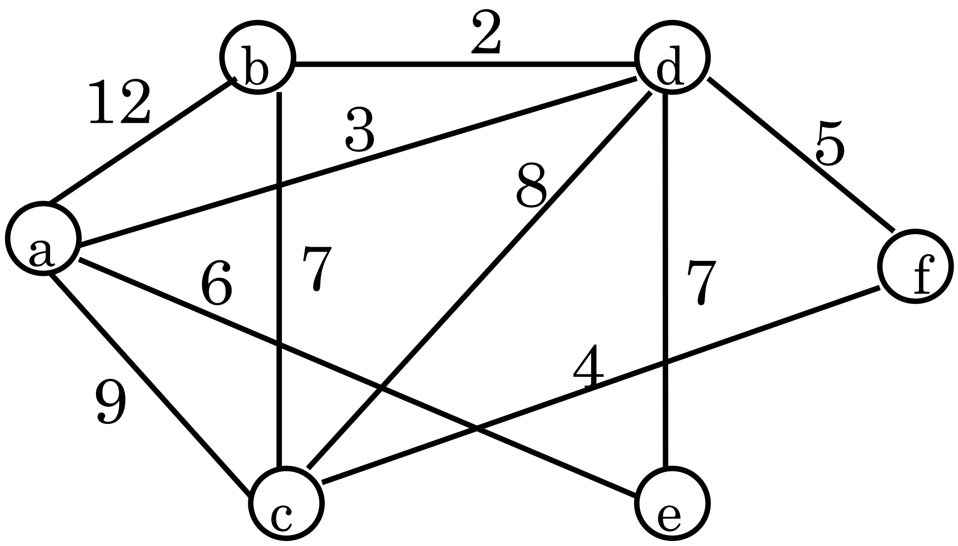
be the sum of weights of edges in $\boldsymbol{T}$, where $\boldsymbol{W(v,w)}$ denotes the weight of edge $\boldsymbol{(v,w)}$.

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

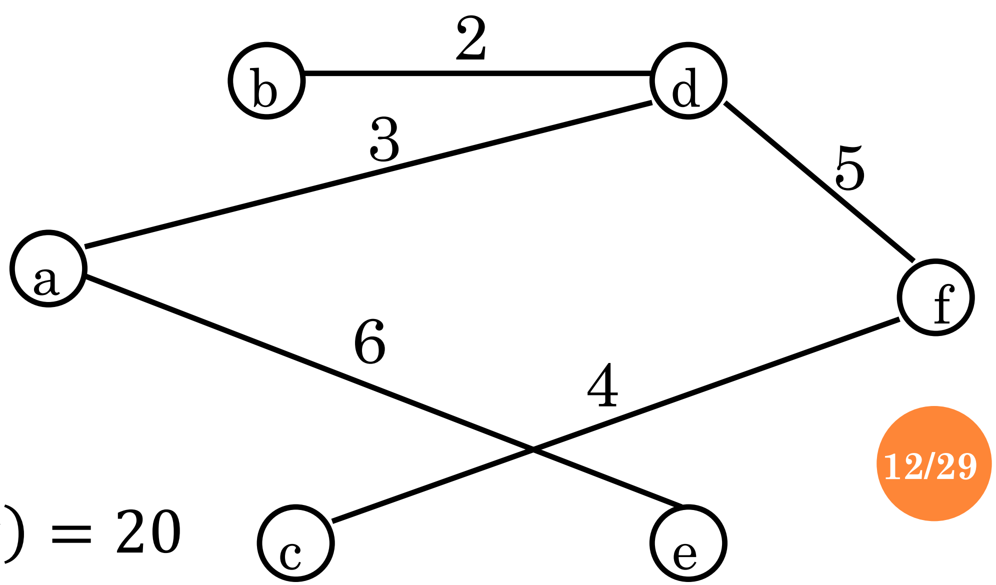
$$\boldsymbol{W(T^m) = \min_{T}\{W(T)\}}$$

MINIMUM SPANNING TREE (MST)

Weighted Graph G



Minimum Spanning Tree of G



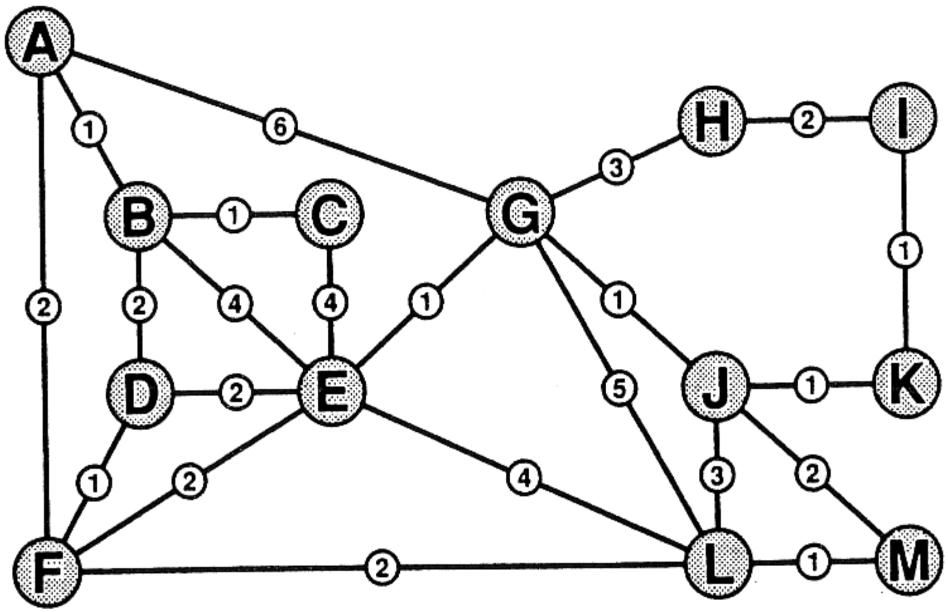
$$W(T^m) = \sum_{(v,w)\in E'} W(v,w) = 20$$

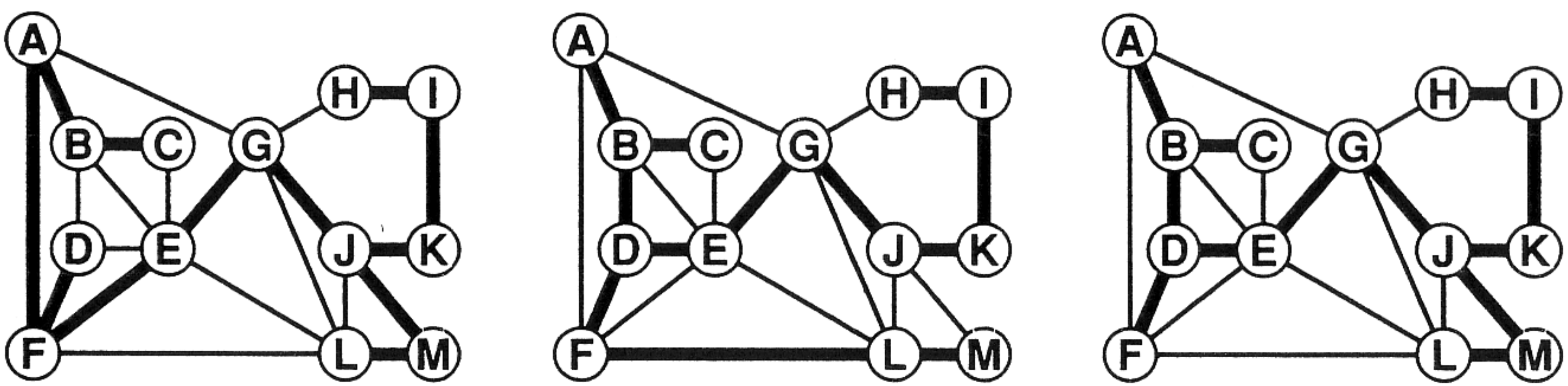
MINIMUM SPANNING TREE (MST)

- **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

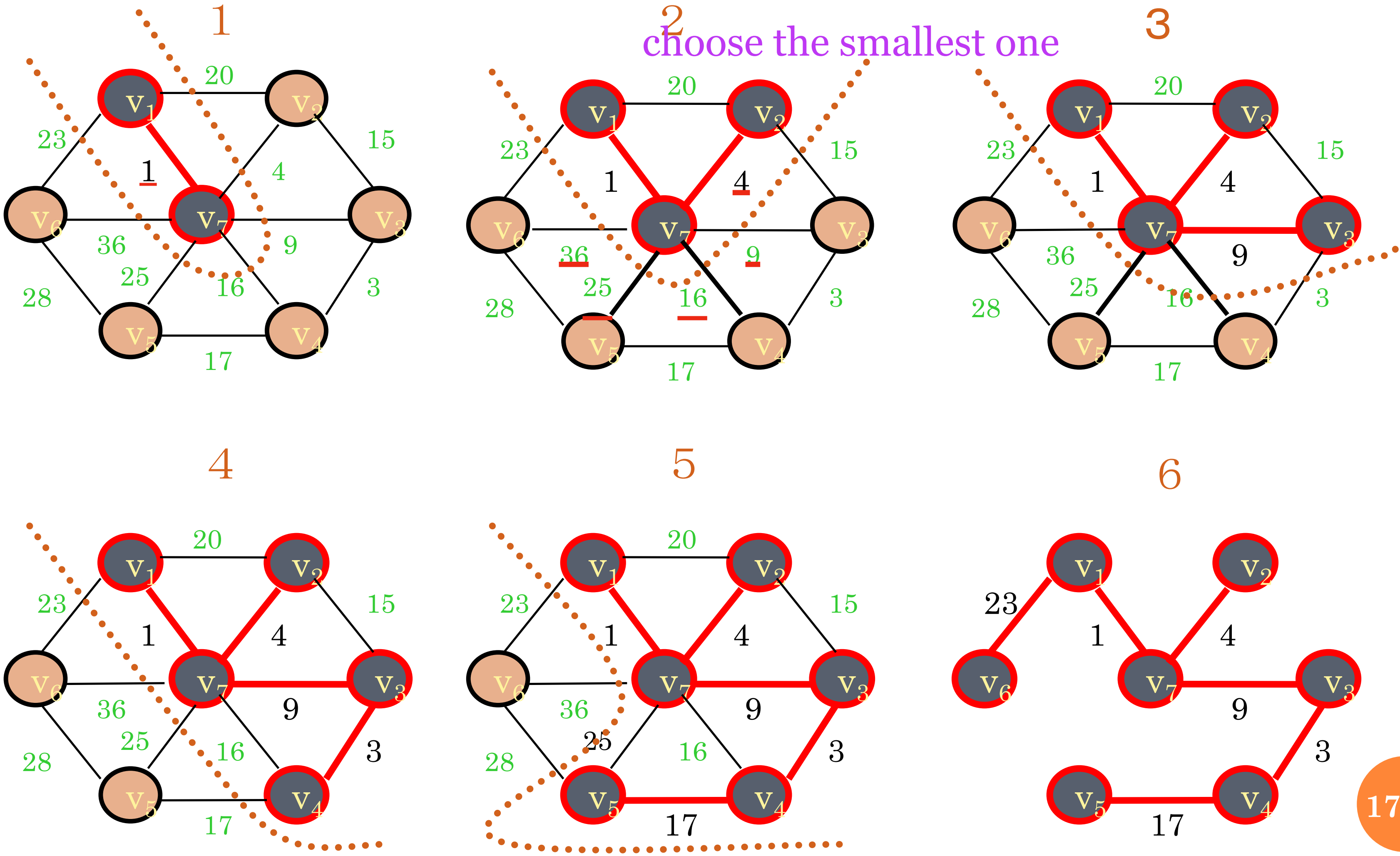
MINIMUM SPANNING TREE (MST)

- **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 $\boldsymbol{T}$.
  - Each vertex is inserted once into and removed once from the priority queue, where each insertion or removal takes $\boldsymbol{O(log\, V)}$ time, in total $\boldsymbol{O(V \log V)}$.
  - The key of a vertex w in the priority queue is modified at most $\boldsymbol{deg(w)}$ times, where each key change takes $\boldsymbol{O(log\, V)}$ time, in total $\boldsymbol{O(E \log V)}$
  - Thus, the time complexity of Prim's algorithm is $\boldsymbol{O(V \log V + E \log V) = O(E \log V)}$.

PRIM'S ALGORITHM

- Implementation:

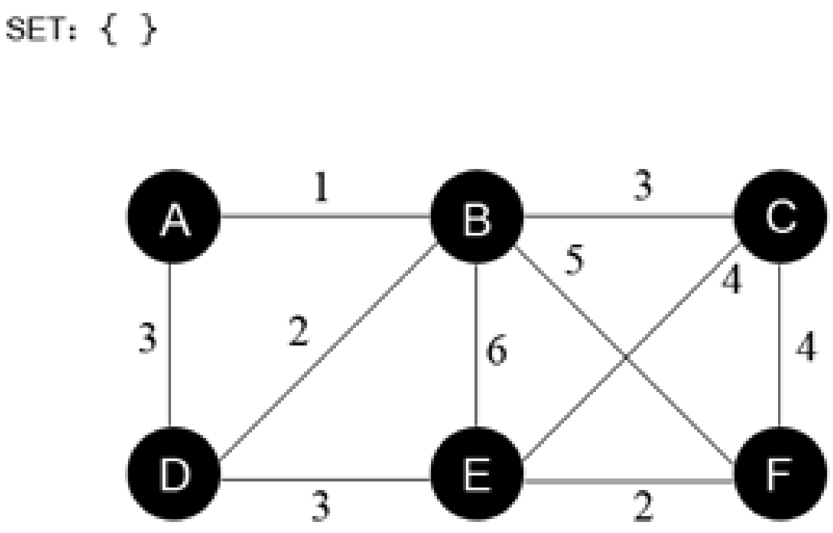
```
def MST-PRIM (G, w, r)
//  Graph G with set of nodes G.V, weight matrix w and
// root node r. MST is the edges set A={(v,v.π), v ∈ V-r}.
    for each u ∈ G.V:
        u.key = ∞
        u.π = NIL
    r.key = 0
    Q = Min-Priority-Queue (G.V)
    while Q ≠ ∅:
        u = Extract-Min (Q)
        for 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 ***v***, the attribute ***v.key*** is the minimum weight of any edge connecting ***v*** to a node in the **MST**.
  - If there is no edge ***v.key*** $= \infty$.
  - The attribute ***v.***$\boldsymbol{\pi}$ names the parent of ***v*** in the **MST**.

PRIM'S ALGORITHM

- Animated example:



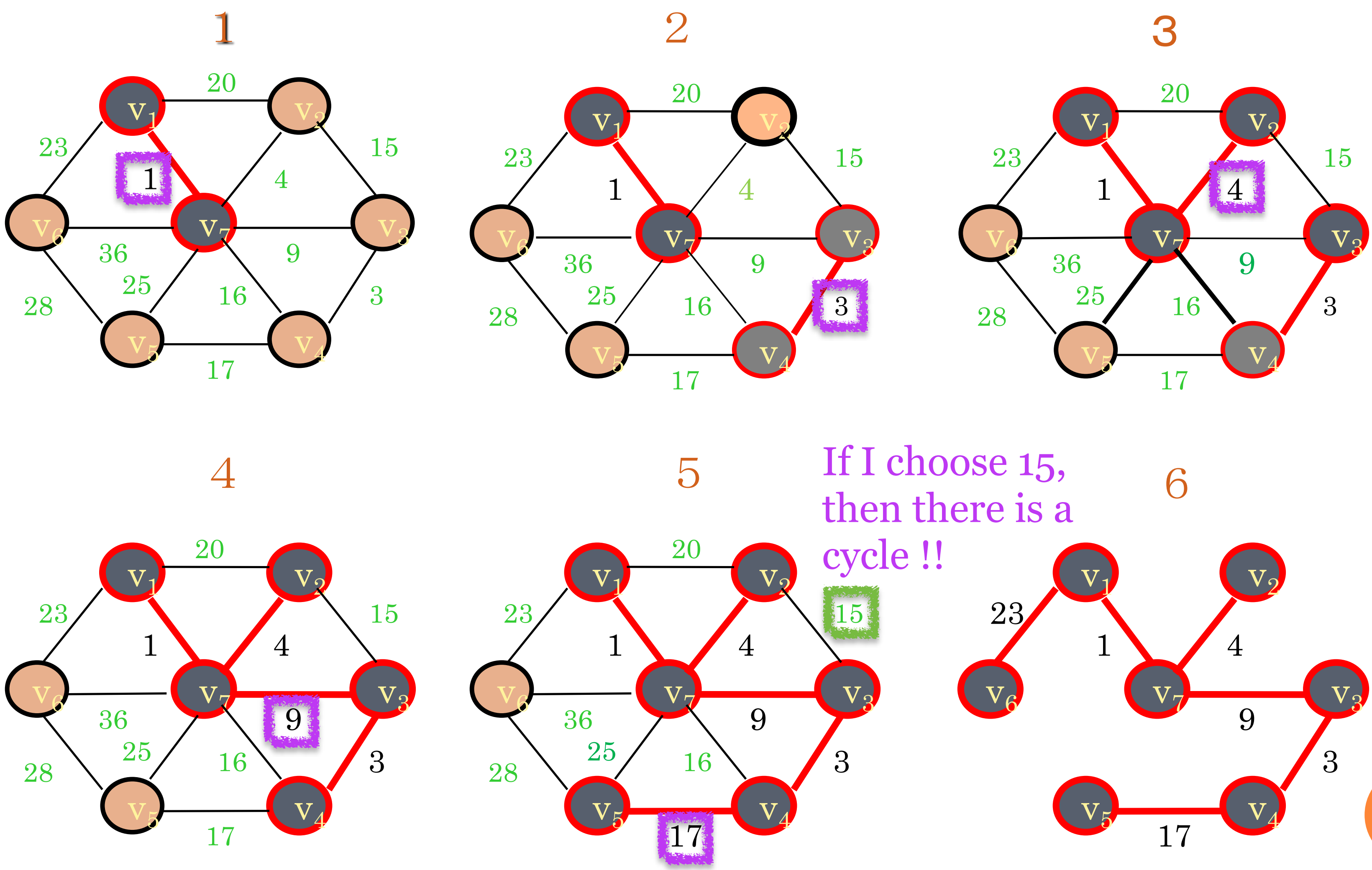
KRUSKAL'S ALGORITHM

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



KRUSKAL'S ALGORITHM

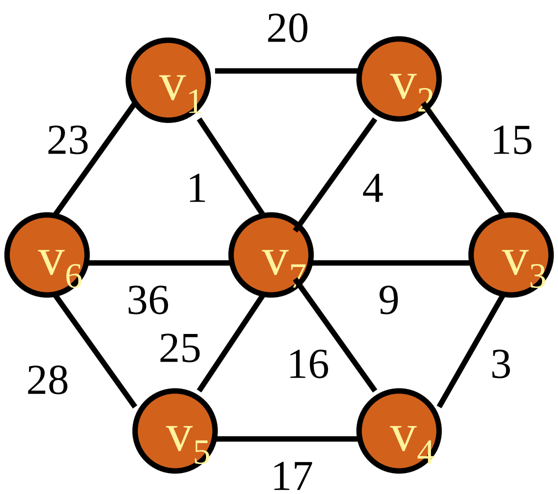
- Implementation:

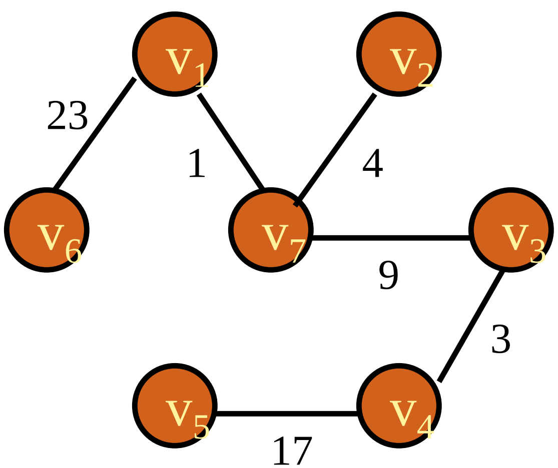
```
def  MST-KRUSKAL (G, w)
//  Graph G with set of nodes G.V, weight matrix w.
//  MST is the edges set A={}.
    A = ∅
    for each v ∈ G.V:
        MAKE-SET (v)
    Sort edges of G.E into non-decreasing order by weight w
    for each edge (u, v) ∈ G.E, taken in non-decreasing order of w:
        if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

KRUSKAL'S ALGORITHM

- Implementation notes.
  - **SET** data structure and operations on it:
    - 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 avoid to make a cycle
    - To make a set from one or several nodes, we use the **MAKE-SET** () operation.

KRUSKAL'S ALGORITHM

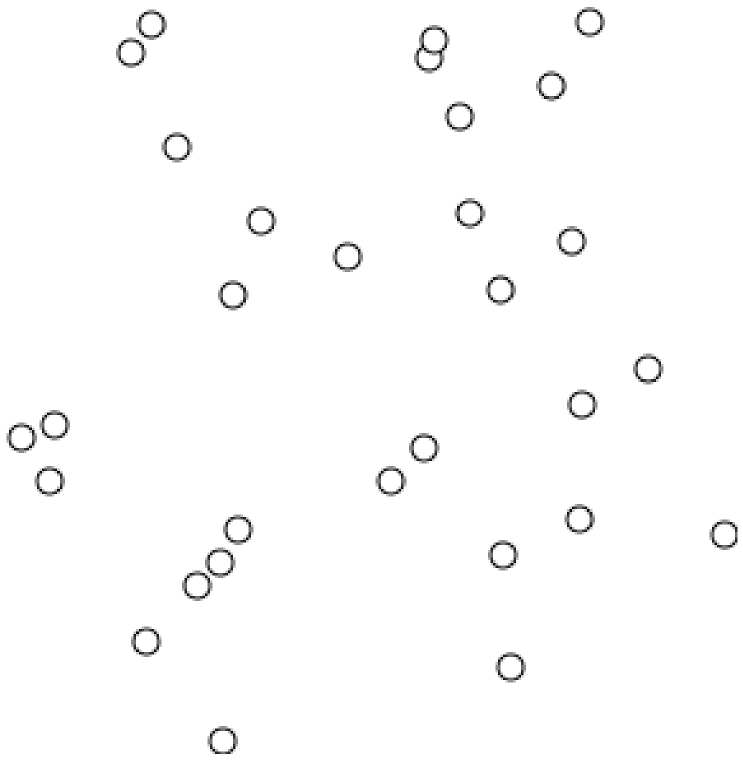




Edge	Action	Sets
		$\{v_1\},\{v_2\},\{v_3\},\{v_4\},\{v_5\},\{v_6\},\{v_7\}$
(v_1, v_7)	Add	$\{v_1,v_7\},\{v_2\},\{v_3\},\{v_4\},\{v_5\},\{v_6\}$
(v_3, v_4)	Add	$\{v_1,v_7\},\{v_2\},\{v_3,v_4\},\{v_5\},\{v_6\}$
(v_2, v_7)	Add	$\{v_1,v_2,v_7\},\{v_3,v_4\},\{v_5\},\{v_6\}$
(v_3, v_7)	Add	$\{v_1,v_2,v_3,v_4,v_7\},\{v_5\},\{v_6\}$
(v_2, v_3)	Reject	already they are in the same set!
(v_4, v_7)	Reject	=> A cycle!
(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 $|\boldsymbol{E}| < |\boldsymbol{V}|^{\mathbf{2}}$, then $\mathbf{log}|\mathbf{E}| < \mathbf{log}\,|\boldsymbol{V}|^{\mathbf{2}} = \mathbf{2}\,\mathbf{log}\,|\boldsymbol{V}|$ which is $\boldsymbol{O}(\mathbf{log}\,\boldsymbol{V})$.
  - Total running time is $\boldsymbol{O}(\boldsymbol{E}\,\mathbf{log}\,\boldsymbol{V})$.

That's All for Today!