Lecture #14
Graph Algorithm (1)

Algorithm

JBNU

Jinhong Jung

In This Lecture

- **Minimum spanning tree**
  - Problem definition
  - Application

- **Algorithms for MST**
  - Kruskal’s algorithm
  - Prim’s algorithm

Outline

- Minimum spanning tree

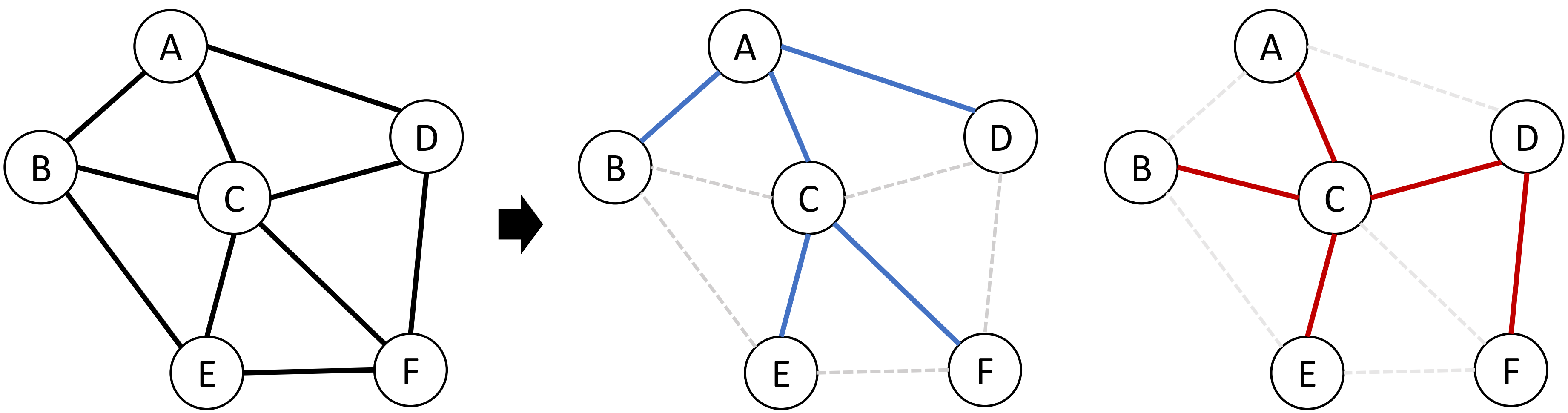
- Kruskal's algorithm

- Prim's algorithm

Spanning Tree

❑ A spanning tree spans the graph like tree

- Acyclic sub-graph including all nodes & connected as tree with $n-1$ edges
  - Suppose the input graph has n nodes
- The graph has a multiple number of spanning trees

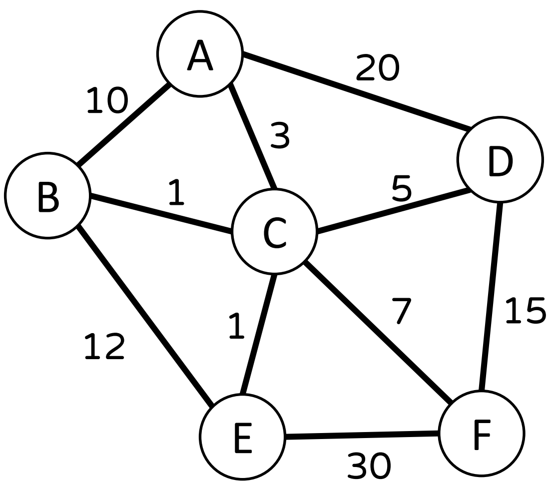


Input graph　　Spanning tree 1　　Spanning tree 2

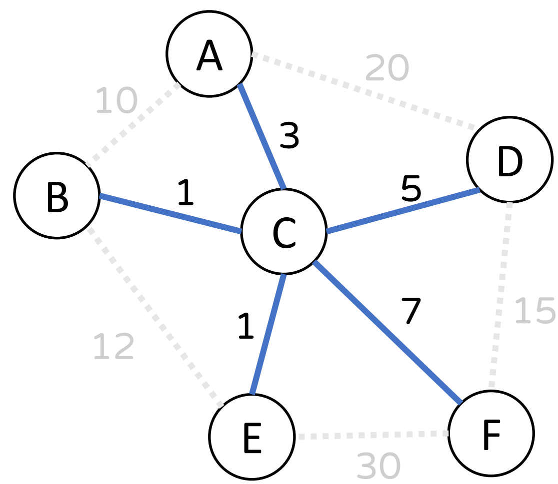
Minimum Spanning Tree

❑ MST = Spanning tree having the minimum cost

- **Input**: a **weighted** and **undirected** graph of n nodes
  - Assuming the graph is **connected**
- **Output**: **minimum cost** of a spanning tree
  - Cost = the sum of edge weights of the tree
- There could be multiple MSTs in one given graph



Input graph

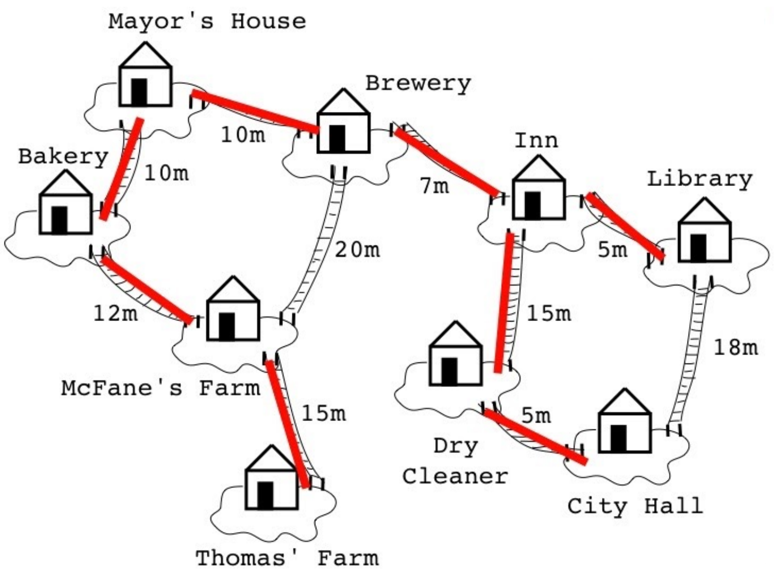


MST (cost = 17)

Why MST?

❑ MST has applications in the design of networks

- Computer networks, telecommunications networks, etc.
- e.g., a telecommunication company tries to install cable along roads in a new neighborhood & suppose installing 1m cable requires 100$
- How can we connect all houses with the minimum expense? ⇒ MST



How To Get MST?

❑ Naïve solution

- Enumerate all possible sub-graphs
- For each sub-graph, check if it is a connected tree and has the minimum cost.

❑ How many sub-graphs are there?

- Count the cases based on whether each edge is included
  - The total sub-graph is 2^m where m is # of edges
- Counting all possible sub-graphs is impractical

❑ Can we quickly solve MST?

- Two algorithms proposed by Kruskal and Prim, resp.

Outline

- ❑ Minimum spanning tree
- ❑ Kruskal's algorithm
- ❑ Prim's algorithm

Kruskal's Strategy

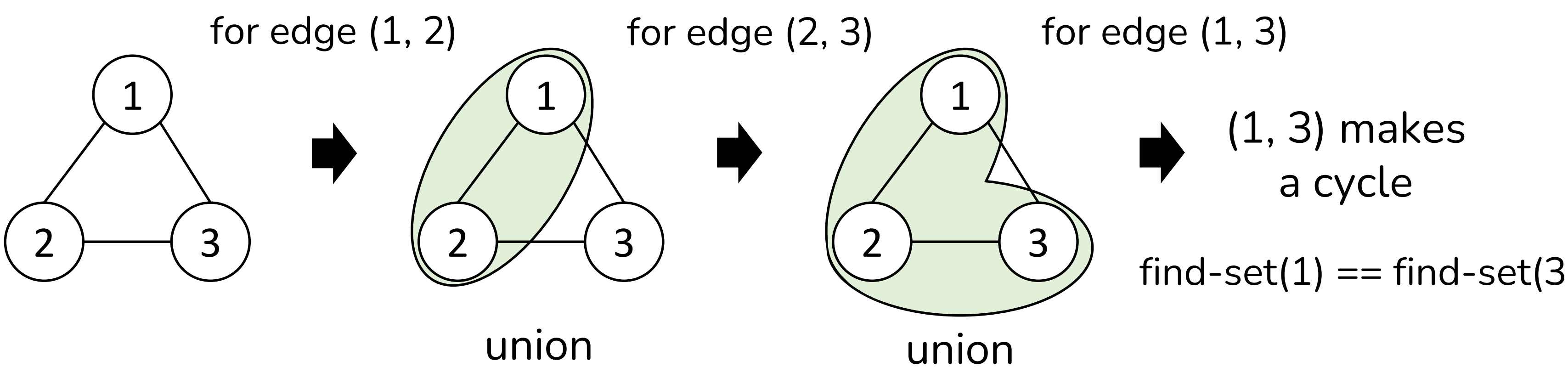
- Incrementally grow MST by adding the minimum edge that does not produce a cycle
  - Step 1. Sort all the edges in increasing order of their weight
  - Step 2. Pick the smallest edge and check if it forms a cycle with the spanning tree formed so far
    - If it doesn't form a cycle, include the edge in MST. Otherwise, discard it.
    - To detect a cycle, we need to use disjoint set!
  - Repeat Step 2 until there are $n - 1$ edges in MST

Joseph Kruskal
1956

How To Detect Cycle

❑ How can we detect a cycle in an undirected graph?

- For each edge (u, v),
  - If u and v are in different sets, then merge their sets (i.e., union(u, v))
  - Else if they are already in the same set, then (u, v) will make a cycle!
  - **Detection condition**: find-set(u) == find-set(v) is true



Kruskal's Algorithm

❑ Psuedocode

```
def kruskal(G):
    T ← list()  # contains edges for MST

    for each u in V:
        make-set(u)

    E′ ← sort the set E of edges of G
         in increasing order of their weights        Step 1

    for each edge (u,v) ∈ E′:
      # if (u,v) does not form a cycle in MST
      if find-set(u) != find-set(v):                 Step 2
          T.add((u,v))
          union(u, v)
```

Example (1)

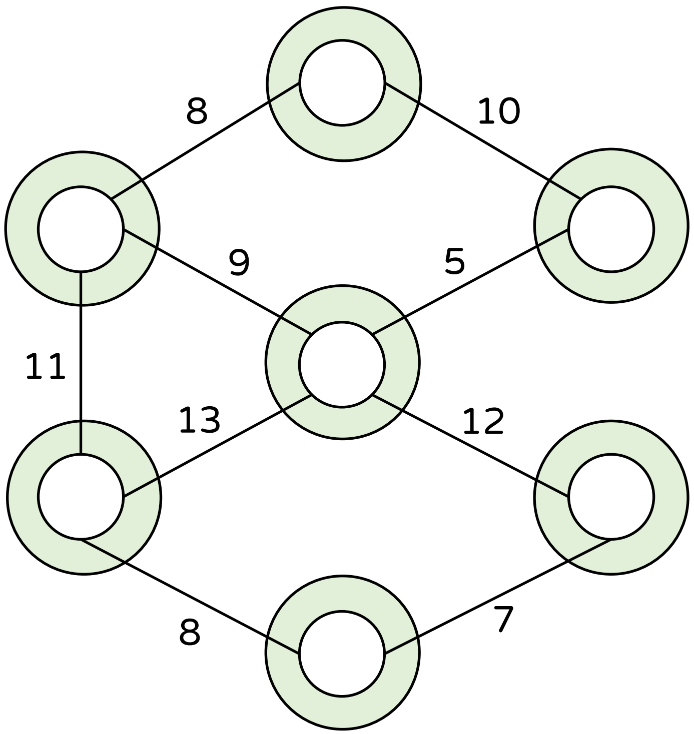
❑ Psuedocode

```
def kruskal(G):
    T ← list()  # contains edges for MST

    for each u in V:
        make-set(u)

    E′ ← sort the set E of edges of G
         in increasing order of their weights

    for each edge (u,v) ∈ E′:
        # if (u,v) does not form a cycle in MST
        if find-set(u) != find-set(v):
            T.add((u,v))
            union(u, v)
```



Example (2)

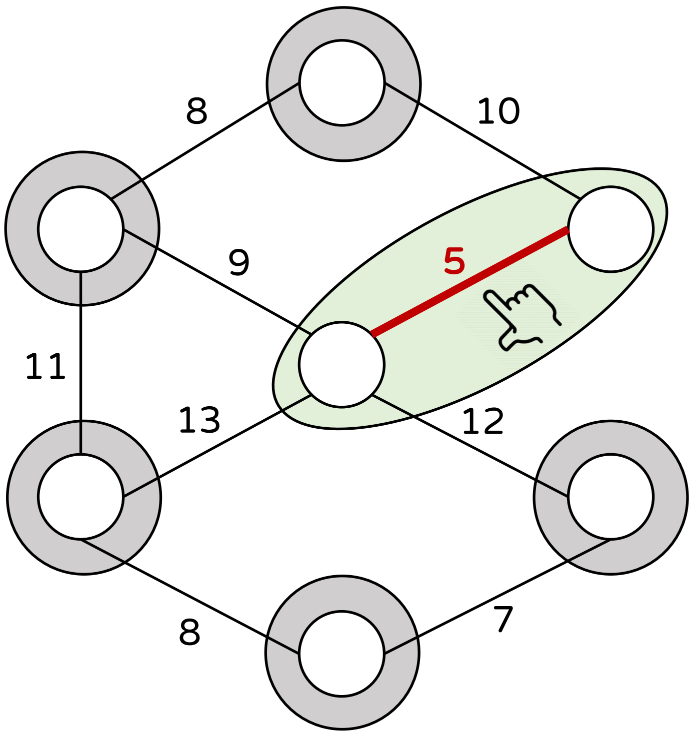
❑ Psuedocode

```
def kruskal(G):
    T ← list()  # contains edges for MST

    for each u in V:
        make-set(u)

    E′ ← sort the set E of edges of G
         in increasing order of their weights

    for each edge (u,v) ∈ E′:
        # if (u,v) does not form a cycle in MST
        if find-set(u) != find-set(v):
            T.add((u,v))
            union(u, v)
```



Example (3)

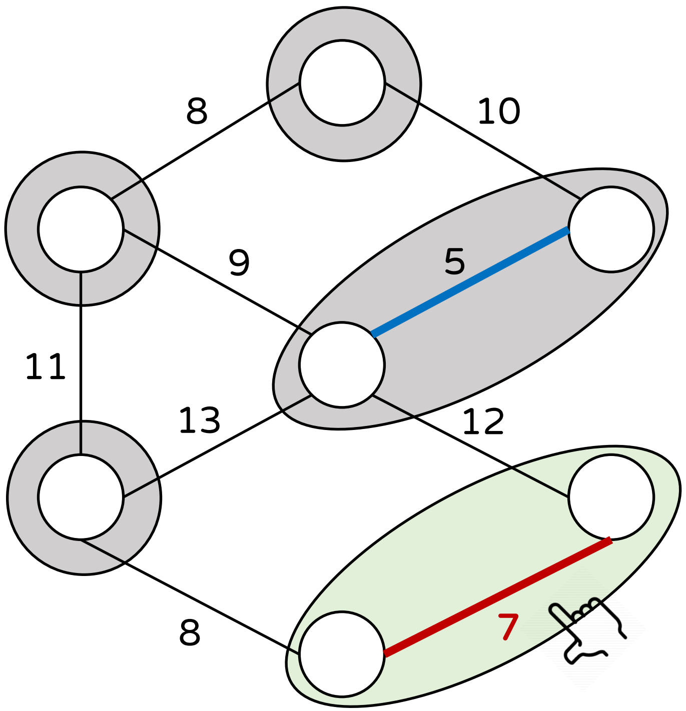
❑ Pseudocode

def kruskal(G):
    $T \leftarrow$ list()   # contains edges for MST

    for each u in V:
        make-set(u)

    $E' \leftarrow$ sort the set E of edges of G
        in increasing order of their weights

    for each edge $(u, v) \in E'$:
        # if (u, v) does not form a cycle in MST
        if find-set(u) != find-set(v):
            T.add((u, v))
            union(u, v)



Example (4)

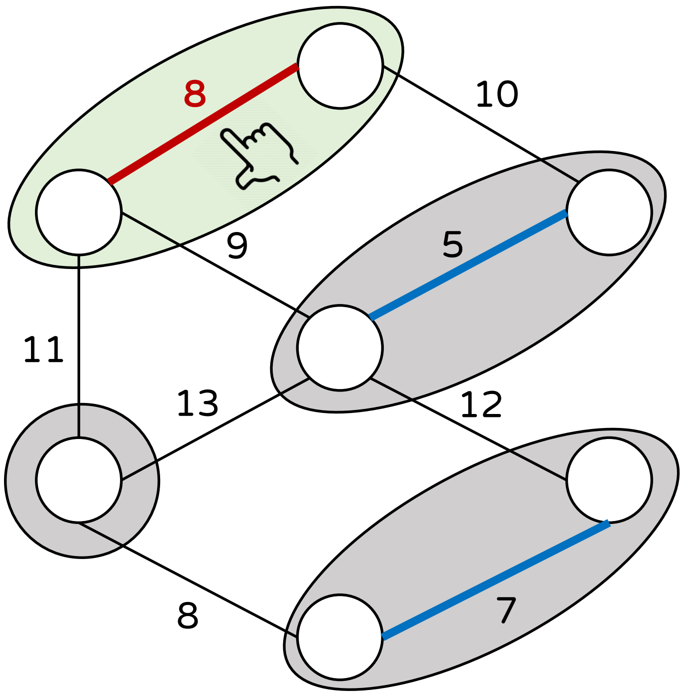
❑ Psuedocode

```
def kruskal(G):
    T ← list()  # contains edges for MST

    for each u in V:
        make-set(u)

    E′ ← sort the set E of edges of G
         in increasing order of their weights

    for each edge (u,v) ∈ E′:
        # if (u,v) does not form a cycle in MST
        if find-set(u) != find-set(v):
            T.add((u,v))
            union(u, v)
```



Example (5)

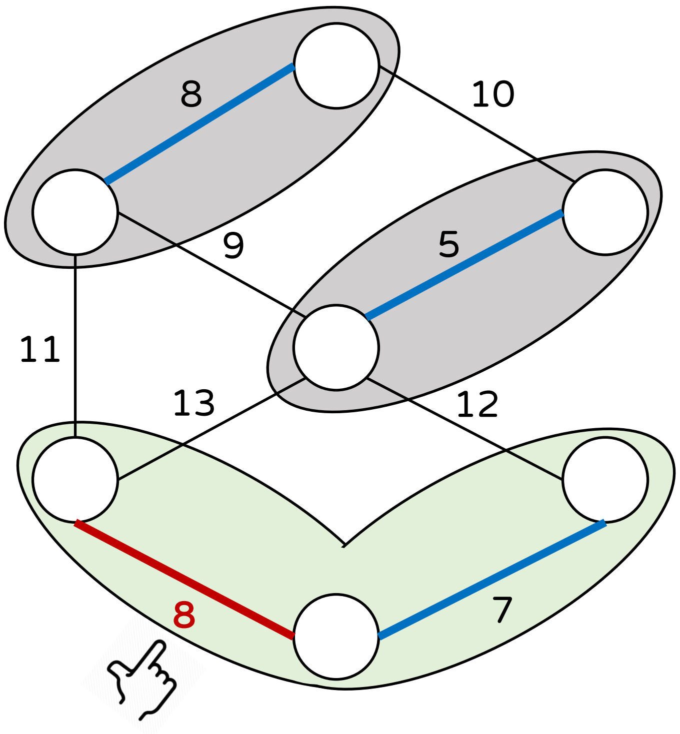
❑ Psuedocode

```
def kruskal(G):
    T ← list()  # contains edges for MST

    for each u in V:
        make-set(u)

    E′ ← sort the set E of edges of G
         in increasing order of their weights

    for each edge (u,v) ∈ E′:
        # if (u,v) does not form a cycle in MST
        if find-set(u) != find-set(v):
            T.add((u,v))
            union(u, v)
```



Example (6)

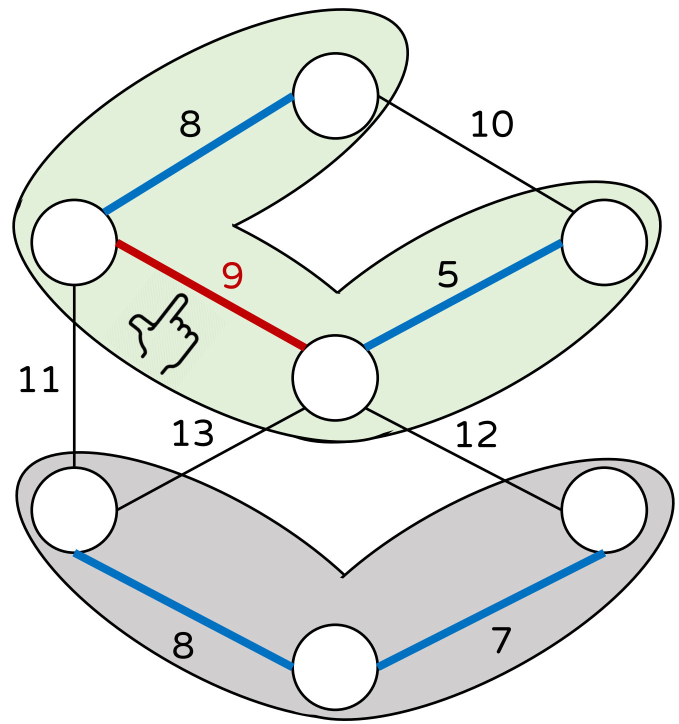
❑ Psuedocode

```
def kruskal(G):
    T ← list()  # contains edges for MST

    for each u in V:
        make-set(u)

    E′ ← sort the set E of edges of G
         in increasing order of their weights

    for each edge (u,v) ∈ E′:
        # if (u,v) does not form a cycle in MST
        if find-set(u) != find-set(v):
            T.add((u,v))
            union(u, v)
```



Example (7)

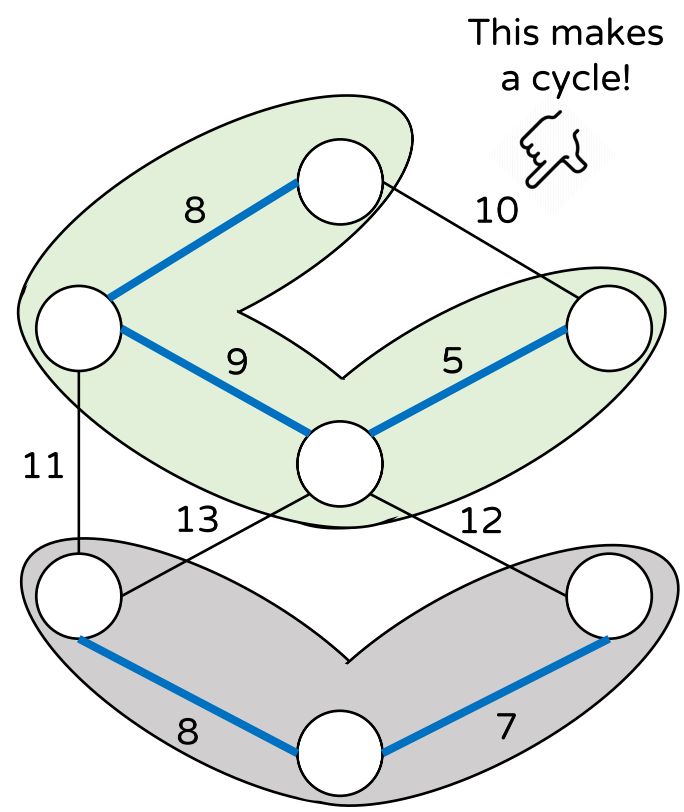
❑ Pseudocode

```
def kruskal(G):
    T ← list()   # contains edges for MST

    for each u in V:
        make-set(u)

    E′ ← sort the set E of edges of G
         in increasing order of their weights

    for each edge (u,v) ∈ E′:
        # if (u,v) does not form a cycle in MST
        if find-set(u) != find-set(v):
            T.add((u,v))
            union(u, v)
```



Example (8)

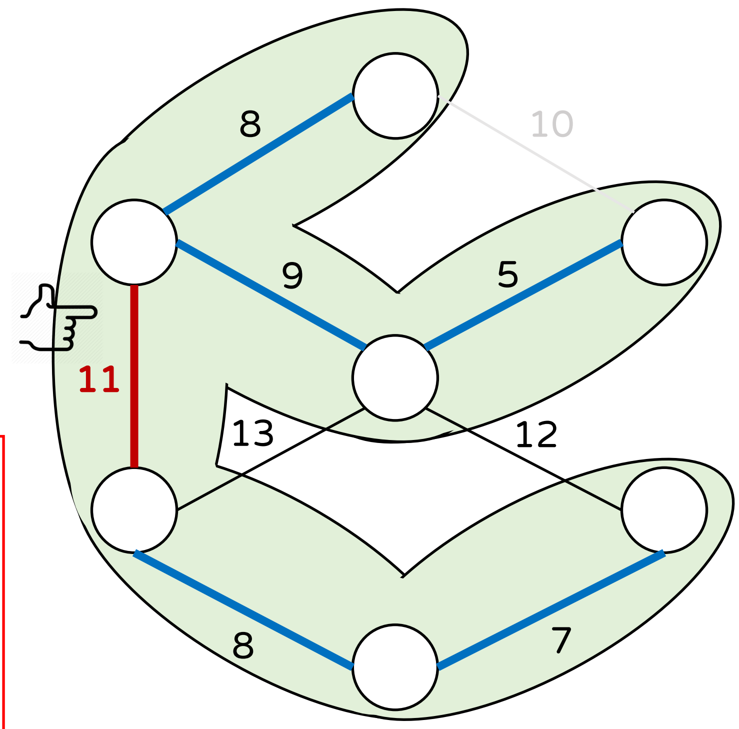
❑ Pseudocode

```
def kruskal(G):
    T ← list()  # contains edges for MST

    for each u in V:
        make-set(u)

    E′ ← sort the set E of edges of G
         in increasing order of their weights

    for each edge (u,v) ∈ E′:
        # if (u,v) does not form a cycle in MST
        if find-set(u) != find-set(v):
            T.add((u,v))
            union(u, v)
```



Example (9)

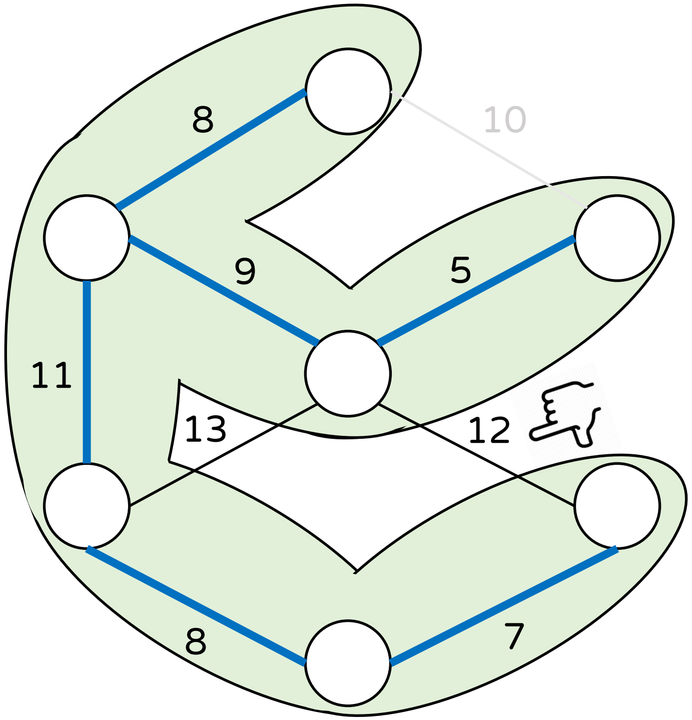
❑ Psuedocode

```
def kruskal(G):
    T ← list()  # contains edges for MST

    for each u in V:
        make-set(u)

    E′ ← sort the set E of edges of G
         in increasing order of their weights

    for each edge (u,v) ∈ E′:
        # if (u,v) does not form a cycle in MST
        if find-set(u) != find-set(v):
            T.add((u,v))
            union(u, v)
```



This makes a cycle

Example (10)

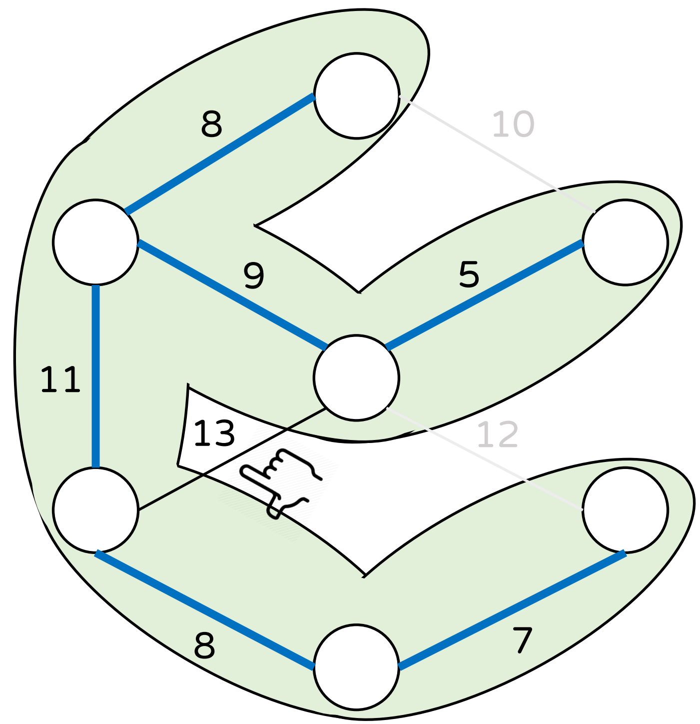
❑ Pseudocode

```
def kruskal(G):
    T ← list()  # contains edges for MST

    for each u in V:
        make-set(u)

    E′ ← sort the set E of edges of G
         in increasing order of their weights

    for each edge (u,v) ∈ E′:
        # if (u,v) does not form a cycle in MST
        if find-set(u) != find-set(v):
            T.add((u,v))
            union(u, v)
```



This makes a cycle

Example (11)

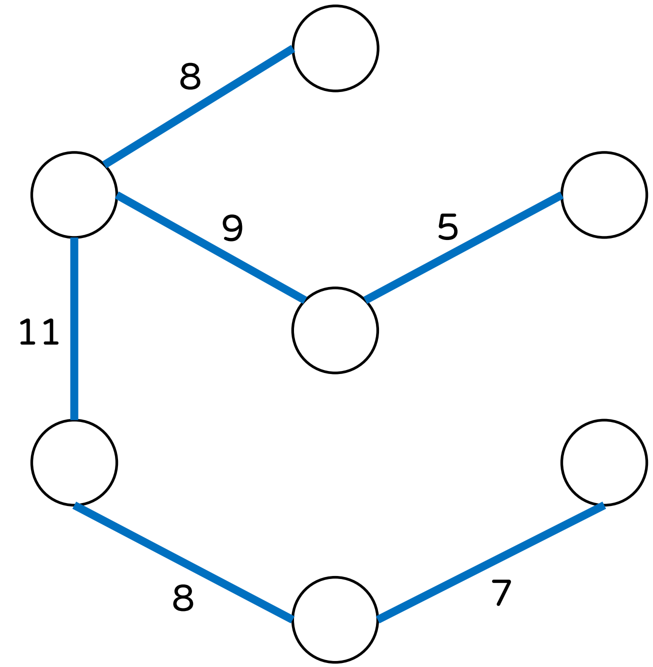
❑ Psuedocode

```
def kruskal(G):
    T ← list()  # contains edges for MST

    for each u in V:
        make-set(u)

    E' ← sort the set E of edges of G
         in increasing order of their weights

    for each edge (u,v) ∈ E':
        # if (u,v) does not form a cycle in MST
        if find-set(u) != find-set(v):
            T.add((u,v))
            union(u, v)
```



Final MST by Kruskal

Complexity Analysis (1)

- **Space complexity:** $O(n+m)$ **space**
  - Adjacency list for G uses $O(n+m)$ space
  - Disjoint set uses $O(n)$ space to store n items
- **Time complexity**
  - Sorting (e.g., heap sort)
    - $O(m \log m)$ time is required to sort m edges (items)
  - Optimized disjoint set: find-set & union of
    - Among P operations consisting of make-set, find-set, and union, let n be the number of make-set operations
    - Then, the time complexity of P operations is $O(P \log^* n)$.
      - $\log^* n = \min\{k \mid \underbrace{\log \log \cdots \log}_{k} n \leq 1\}$

Complexity Analysis (2)

❑ Time complexity: $O(m \log n)$

- m is # of edges and n is # of nodes

```
def kruskal(G):
    T ← list()  # contains edges for MST

    for each u in V:
        make-set(u)

    E' ← sort the set E of edges of G
         in increasing order of their weights

    for each edge (u,v) ∈ E':
        # if (u,v) does not form a cycle in MST
        if find-set(u) != find-set(v):
            T.add((u,v))
            union(u, v)
```

make-set: n times

$m \log m \leq m \log n^2 = O(m \log n)$

find-set: $2m$ times
union: $n - 1$ times
$\Rightarrow P = 2n + 2m - 1$
$\Rightarrow O(P \log^* n)$
$\Rightarrow O(n + m)$
where $\log^* n$ is pratically constant

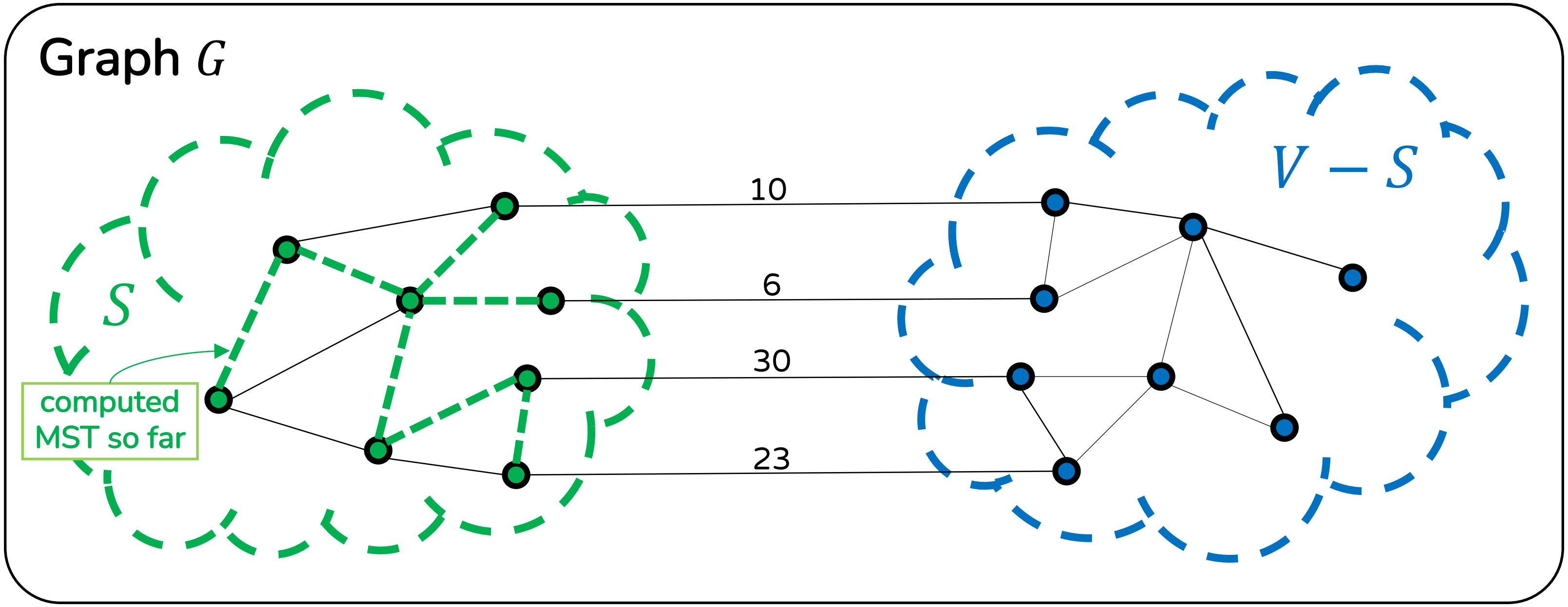
Outline

- ❑ Minimum spanning tree
- ❑ Kruskal's algorithm
- ❑ Prim's algorithm

Prim's Strategy (1)

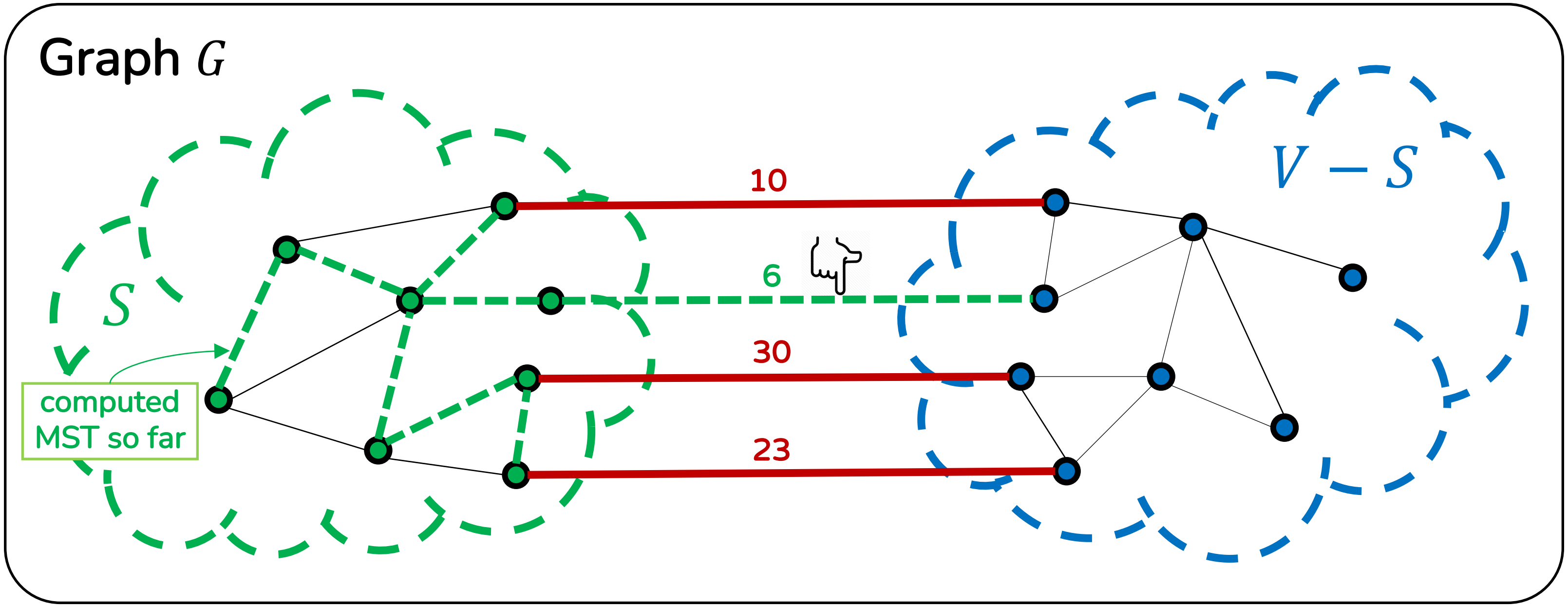
Robert Prim
1957

- **Incrementally grow MST by adding a new node connected with a minimum crossing edge**
  - **Step 1.** maintain two sets S and $V - S$
    - S is a set of nodes consisting of the current MST
    - $V - S$ is a set of remaining nodes where V is set of nodes in G



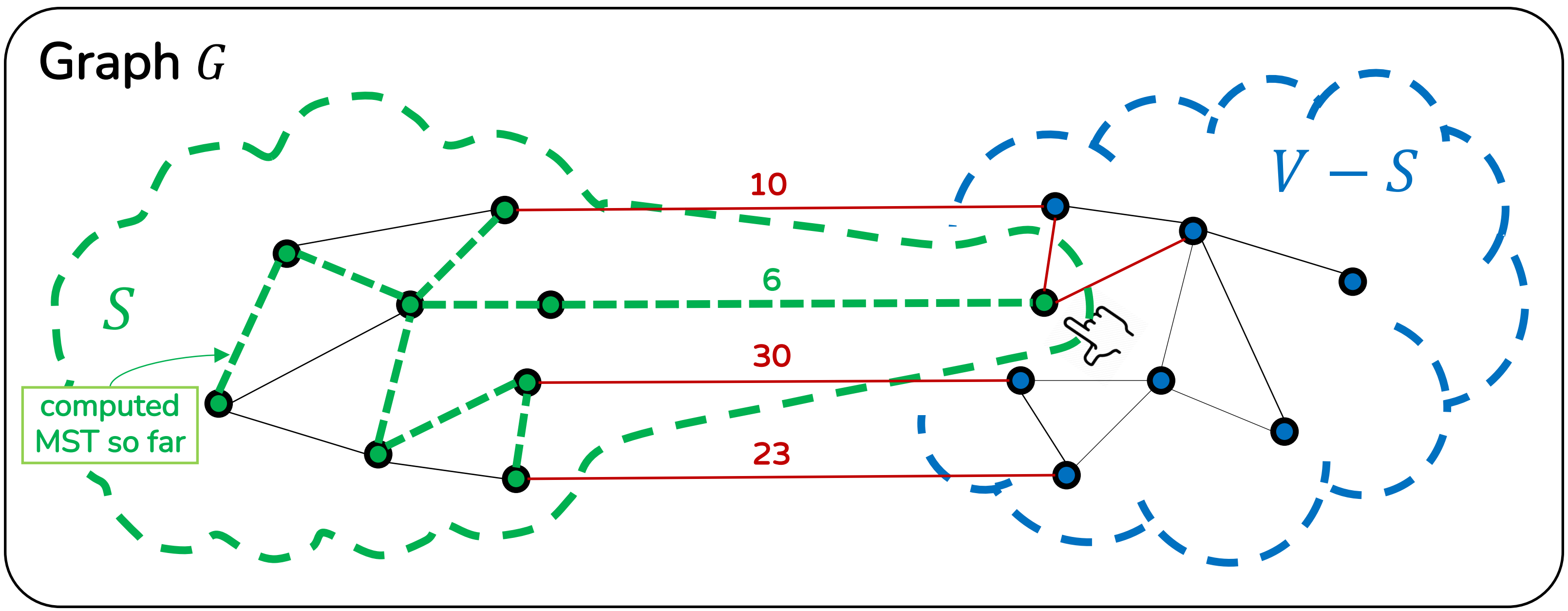
Prim's Strategy (2)

- Incrementally grow MST by adding a new node connected with a minimum crossing edge
  - **Step 2.** among all crossing edges that connects two sets, **select the minimum edge.**
    - Below we select the edge of weight 6 which is the minimum



Prim’s Strategy (3)

- Incrementally grow MST by adding a new node connected with a minimum crossing edge
  - **Step 2** leads to moving the other endpoint of the edge into S
  - Repeat Step 2 until S becomes V



Prim's Algorithm

❑ Psuedocode

- r: initiating node for finding MST (any node can initiate)

- If $v \in S$, $c[v]$ is the min. weight of **the crossing edge (*, v) selected by Prim**.
- If $v \in V - S$, $c[v]$ is the smallest weight of **crossing edges (*, v) checked so far.**

```
def prim(G, r):
    S ← ∅
    for each v in V:                         # initialization (step 1)
        c[v] ← ∞
    c[r] ← 0

    while S is not V:
        u ← extract-min(V − S, c)            # argmin_{u∈V−S} c[u]
        S ← S ∪ {u}                          # u is included into MST (step 2)
        for each v in N_u:                   # for each edge (u, v)
            if v ∈ V − S and w(u,v) < c[v]:  # if (u, v) is crossing & c[v] is updatable (relaxable)
                c[v] ← w(u,v)                # update (relax) smaller weight of (u, v) on c[v]
                parent[v] ← u                # put a trace for (u, v)
```

Update for the next stage

Example (1)

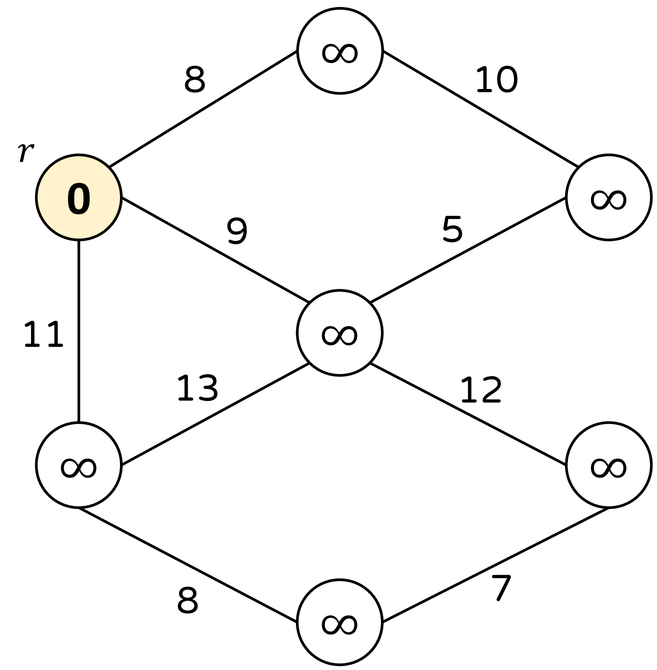
❑ Psuedocode

Value in each circle is $c[]$

```
def prim(G, r):
    S ← ∅
    for each v in V:
        c[v] ← ∞
    c[r] ← 0

    while S is not V:
        u ← extract-min(V − S, c)
        S ← S ∪ {u}
        for each v in N_u:
            if v ∈ V − S and w(u,v) < c[v]:
                c[v] ← w(u,v)
                parent[v] ← u
```

initialization



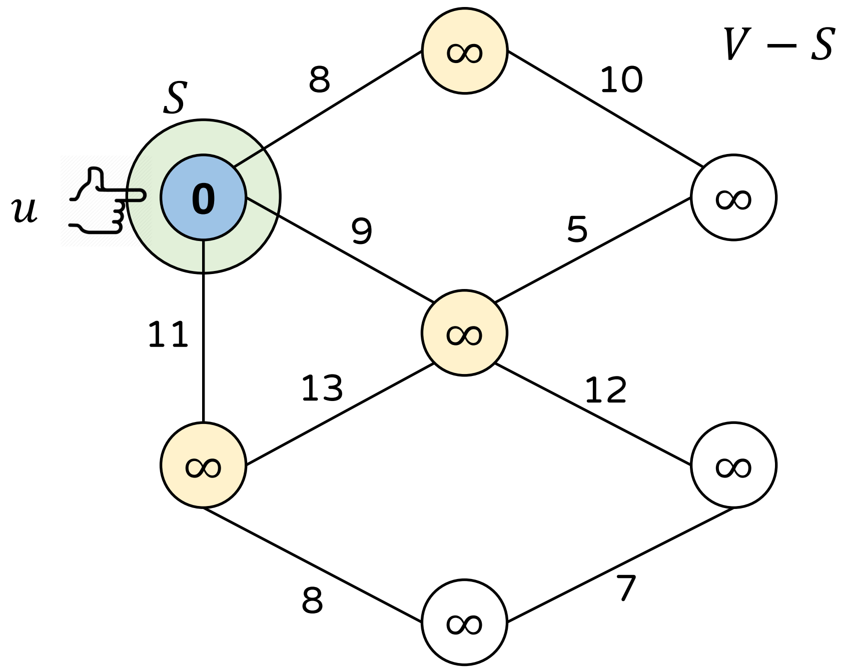
Example (2)

❑ Pseudocode

Value in each circle is $c[]$

```
def prim(G, r):
    S ← ∅
    for each v in V:
        c[v] ← ∞
    c[r] ← 0

    while S is not V:
        u ← extract-min(V − S, c)
        S ← S ∪ {u}
        for each v in N_u:
            if v ∈ V − S and w(u,v) < c[v]:
                c[v] ← w(u,v)
                parent[v] ← u
```



Example (3)

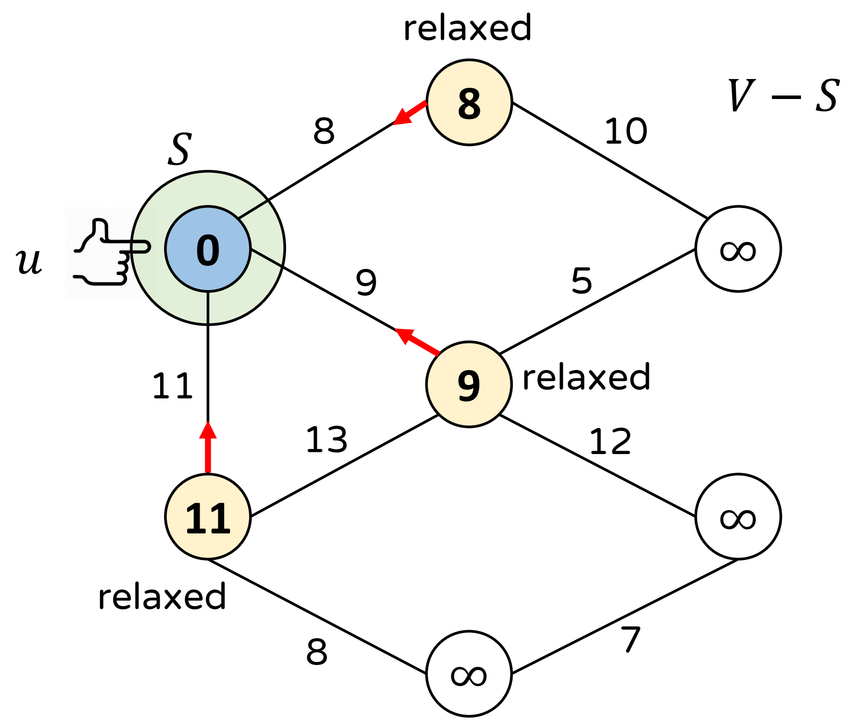
❑ Pseudocode

```
def prim(G, r):
    S ← ∅
    for each v in V:
        c[v] ← ∞
    c[r] ← 0

    while S is not V:
        u ← extract-min(V − S, c)
        S ← S ∪ {u}
        for each v in N_u:
            if v ∈ V − S and w(u,v) < c[v]:
                c[v] ← w(u,v)
                parent[v] ← u
```

Value in each circle: $c[]$

$u \leftarrow v$: a trace that `parent[v] ← u`



Example (4)

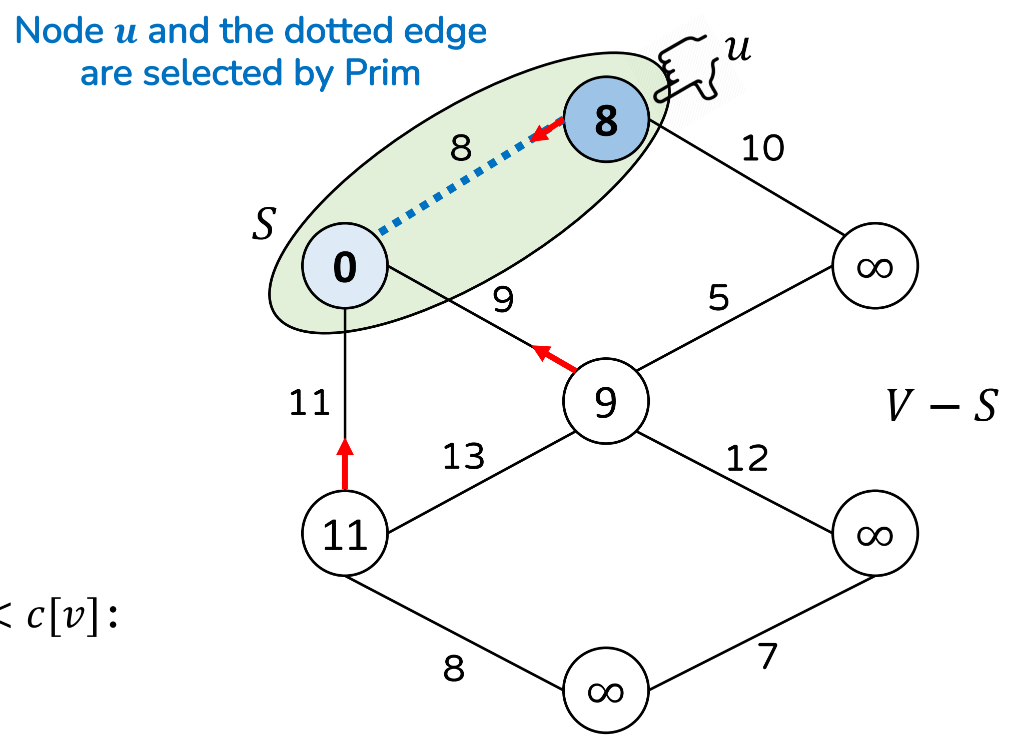
❑ Psuedocode

Value in each circle: $c[]$

$u \leftarrow v$: a trace that `parent[v] ← u`

```
def prim(G, r):
    S ← ∅
    for each v in V:
        c[v] ← ∞
    c[r] ← 0

    while S is not V:
        u ← extract-min(V − S, c)
        S ← S ∪ {u}
        for each v in N_u:
            if v ∈ V − S and w(u,v) < c[v]:
                c[v] ← w(u,v)
                parent[v] ← u
```



Example (5)

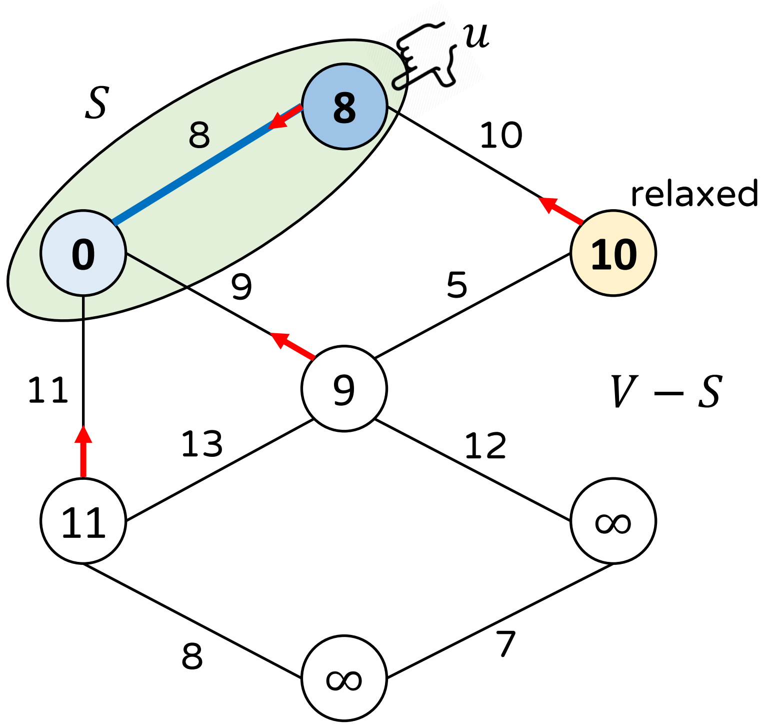
❑ Psuedocode

```
def prim(G, r):
    S ← ∅
    for each u in V:
        c[u] ← ∞
    c[r] ← 0

    while S is not V:
        u ← extract-min(V − S, c)
        S ← S ∪ {u}
        for each v in N_u:
            if v ∈ V − S and w(u,v) < c[v]:
                c[v] ← w(u,v)
                parent[v] ← u
```

Value in each circle: $c[]$

$u \leftarrow v$: a trace that `parent[v] ← u`



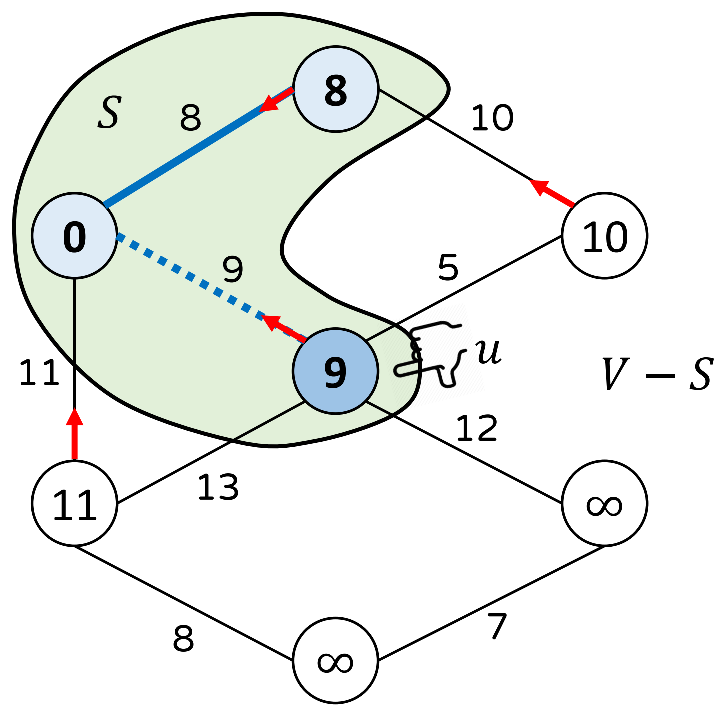
Example (6)

❑ Psuedocode

Value in each circle: $c[]$
$u \leftarrow v$: a trace that `parent[v] ← u`

```
def prim(G, r):
    S ← ∅
    for each v in V:
        c[v] ← ∞
    c[r] ← 0

    while S is not V:
        u ← extract-min(V − S, c)
        S ← S ∪ {u}
        for each v in N_u:
            if v ∈ V − S and w(u,v) < c[v]:
                c[v] ← w(u,v)
                parent[v] ← u
```



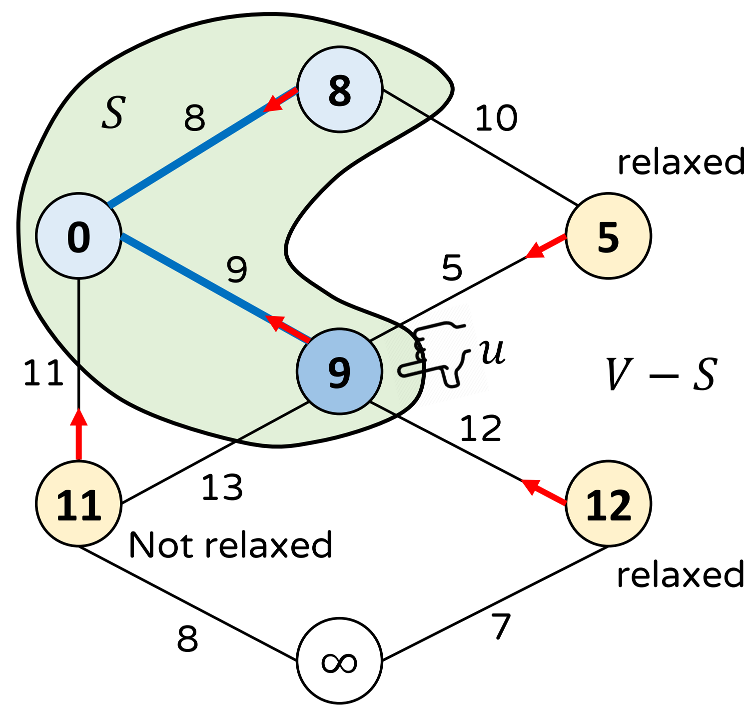
Example (7)

❑ Pseudocode

Value in each circle: $c[]$
$u \leftarrow v$: a trace that `parent[v] ← u`

```
def prim(G, r):
    S ← ∅
    for each v in V:
        c[v] ← ∞
    c[r] ← 0

    while S is not V:
        u ← extract-min(V − S, c)
        S ← S ∪ {u}
        for each v in N_u:
            if v ∈ V − S and w(u,v) < c[v]:
                c[v] ← w(u,v)
                parent[v] ← u
```



Example (8)

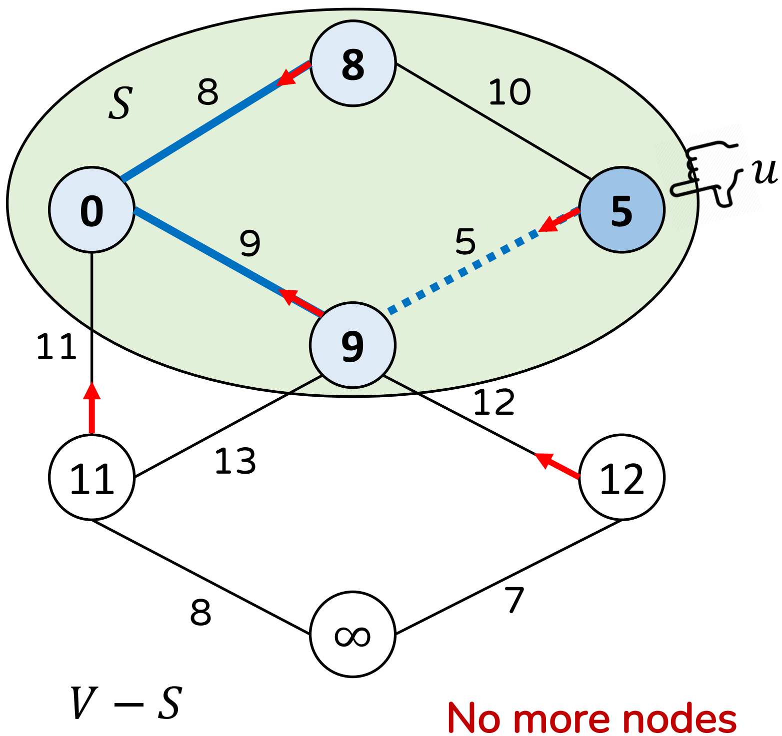
❑ Pseudocode

```
def prim(G, r):
    S ← ∅
    for each v in V:
        c[v] ← ∞
    c[r] ← 0

    while S is not V:
        u ← extract-min(V − S, c)
        S ← S ∪ {u}
        for each v in N_u:
            if v ∈ V − S and w(u,v) < c[v]:
                c[v] ← w(u,v)
                parent[v] ← u
```

Value in each circle: $c[]$

$u \leftarrow v$: a trace that `parent[v] ← u`



No more nodes to be checked from u

Example (9)

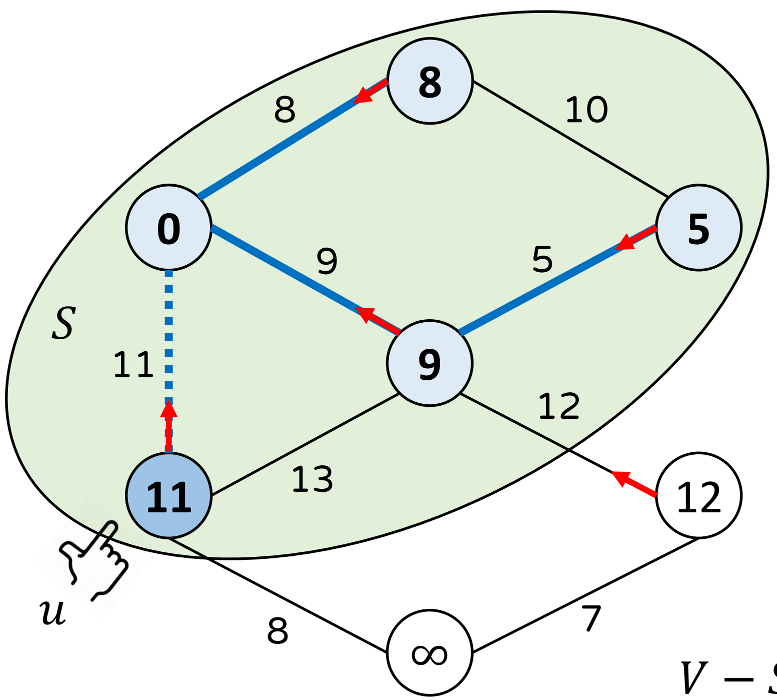
❑ Psuedocode

```
def prim(G, r):
    S ← ∅
    for each v in V:
        c[v] ← ∞
    c[r] ← 0

    while S is not V:
        u ← extract-min(V − S, c)
        S ← S ∪ {u}
        for each v in N_u:
            if v ∈ V − S and w(u,v) < c[v]:
                c[v] ← w(u,v)
                parent[v] ← u
```

Value in each circle: $c[]$

$u \leftarrow v$: a trace that `parent[v] ← u`



Example (10)

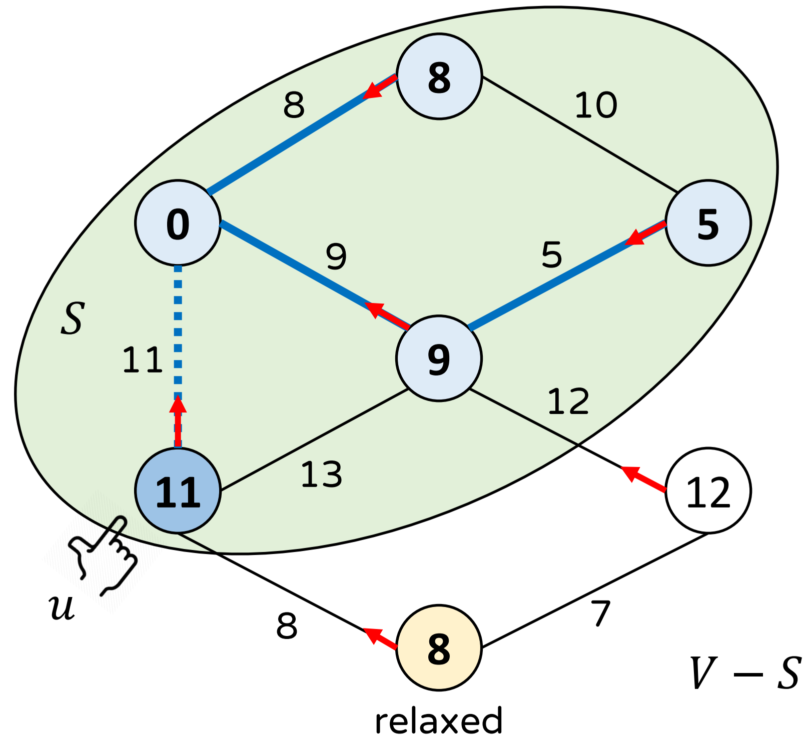
❑ Psuedocode

```
def prim(G, r):
    S ← ∅
    for each v in V:
        c[v] ← ∞
    c[r] ← 0

    while S is not V:
        u ← extract-min(V − S, c)
        S ← S ∪ {u}
        for each v in N_u:
            if v ∈ V − S and w(u,v) < c[v]:
                c[v] ← w(u,v)
                parent[v] ← u
```

Value in each circle: $c[]$

$u \leftarrow v$: a trace that `parent[v] ← u`



Example (11)

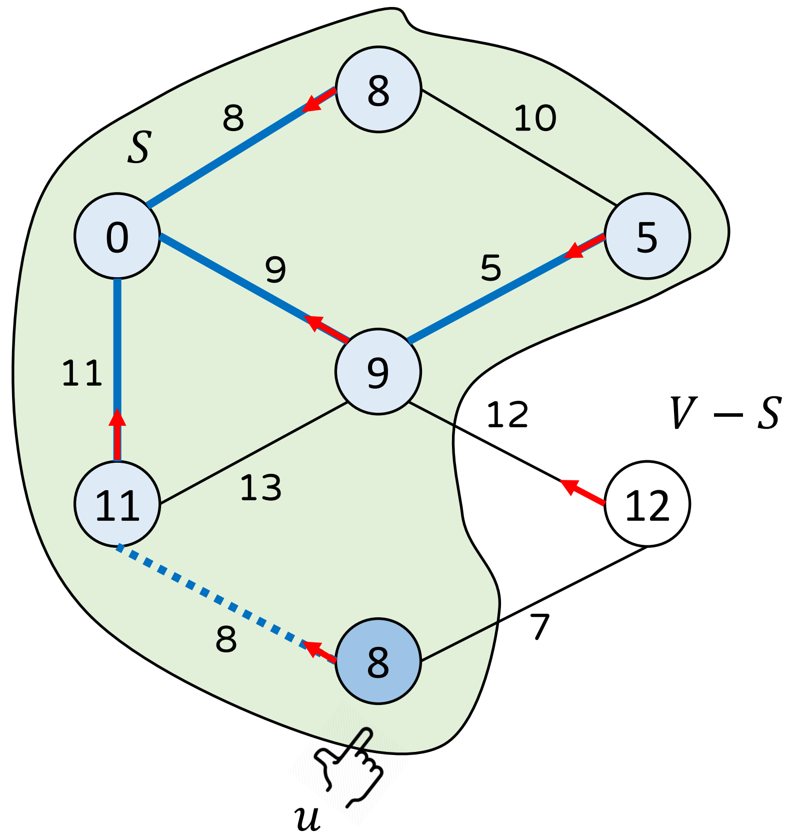
❑ Psuedocode

```
def prim(G, r):
    S ← ∅
    for each v in V:
        c[v] ← ∞
    c[r] ← 0

    while S is not V:
        u ← extract-min(V − S, c)
        S ← S ∪ {u}
        for each v in N_u:
            if v ∈ V − S and w(u,v) < c[v]:
                c[v] ← w(u,v)
                parent[v] ← u
```

Value in each circle: $c[]$

$u \leftarrow v$: a trace that `parent[v] ← u`



Example (12)

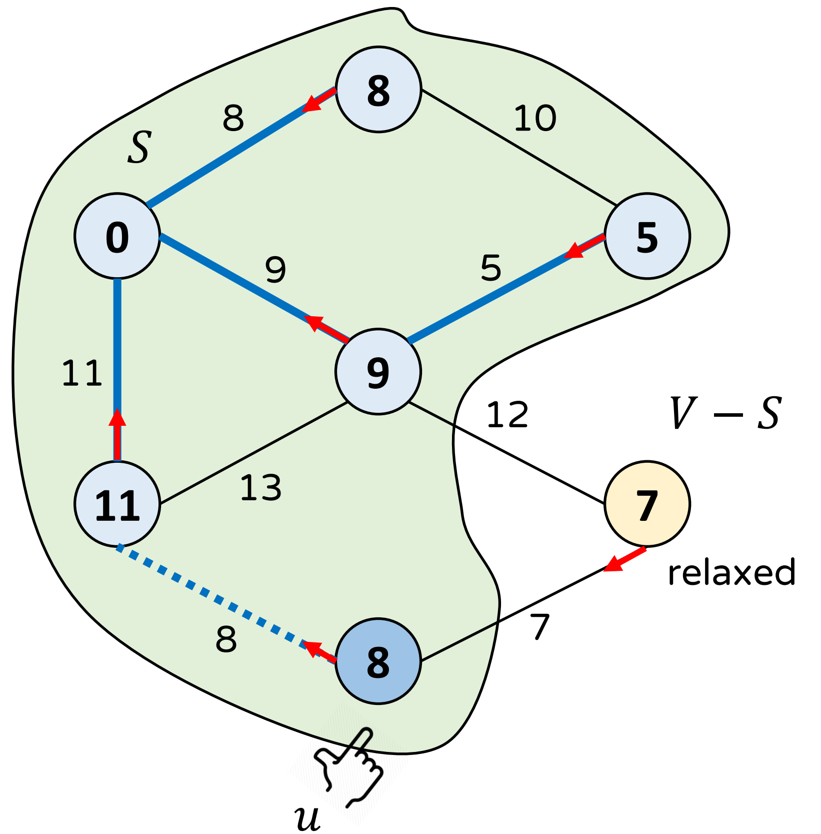
❑ Pseudocode

```
def prim(G, r):
    S ← ∅
    for each v in V:
        c[v] ← ∞
    c[r] ← 0

    while S is not V:
        u ← extract-min(V − S, c)
        S ← S ∪ {u}
        for each v in N_u:
            if v ∈ V − S and w(u,v) < c[v]:
                c[v] ← w(u,v)
                parent[v] ← u
```

Value in each circle: $c[]$

$u \leftarrow v$: a trace that `parent[v] ← u`



Example (13)

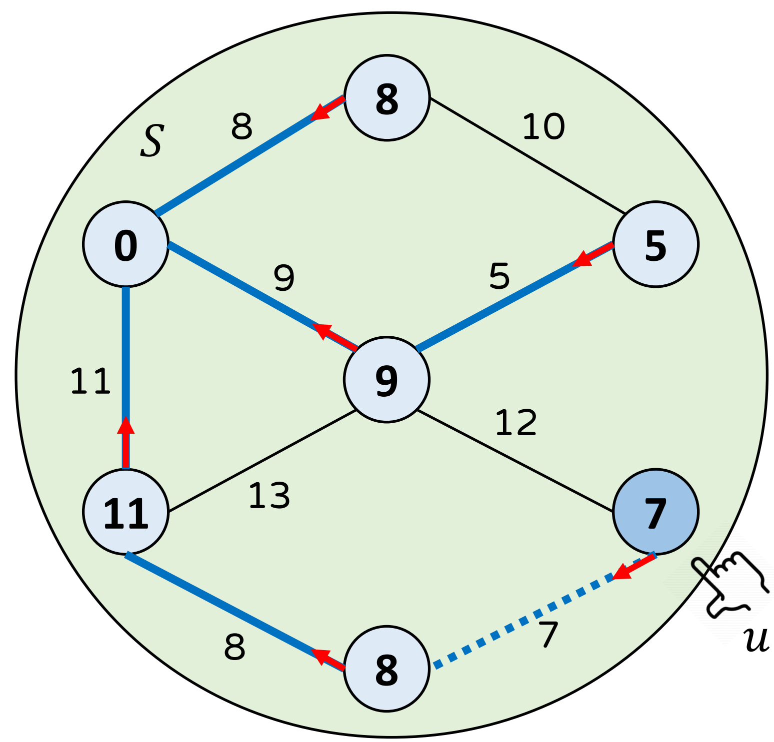
❑ Pseudocode

```
def prim(G, r):
    S ← ∅
    for each v in V:
        c[v] ← ∞
    c[r] ← 0

    while S is not V:
        u ← extract-min(V − S, c)
        S ← S ∪ {u}
        for each v in N_u:
            if v ∈ V − S and w(u,v) < c[v]:
                c[v] ← w(u,v)
                parent[v] ← u
```

Value in each circle: $c[]$

$u \leftarrow v$: a trace that `parent[v] ← u`



Now S becomes V

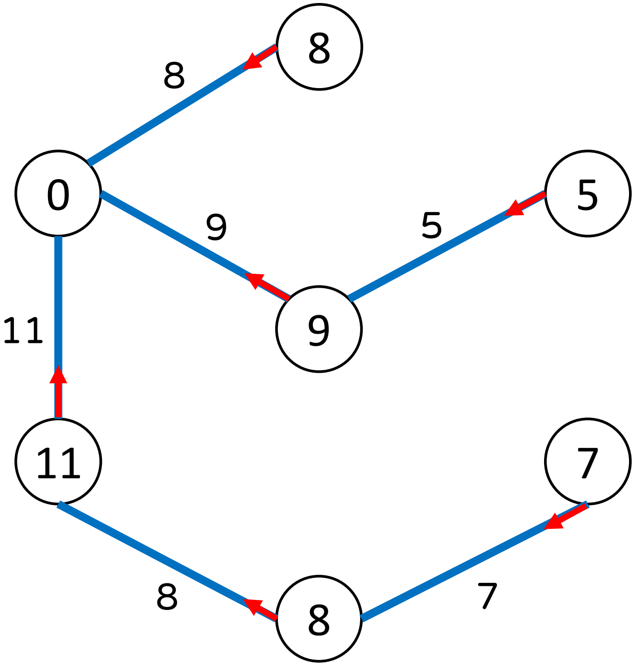
Example (14)

❑ Psuedocode

Value in each circle: $c[]$
$u \leftarrow v$: a trace that `parent[v] ← u`

```
def prim(G, r):
    S ← ∅
    for each v in V:
        c[v] ← ∞
    c[r] ← 0

    while S is not V:
        u ← extract-min(V − S, c)
        S ← S ∪ {u}
        for each v in N_u:
            if v ∈ V − S and w(u,v) < c[v]:
                c[v] ← w(u,v)
                parent[v] ← u
```



Final MST by Prim

Implementation Details (1)

❑ Do not need to track S explicitly

- Let $R = V - S$ denote the set of remining nodes
  - delete-min(R, c[]) $:= v^* = \operatorname{argmin}_{v \in R} c[v]$ and $R \leftarrow R - \{v^*\}$

```
def prim(G, r):
    S ← ∅
    for each v in V:
        c[v] ← ∞
    c[r] ← 0

    while S is not V:
        u ← extract-min(V − S, c)
        S ← S ∪ {u}
        for each v in N_u:
            if v ∈ V − S and w(u,v) < c[v]:
                c[v] ← w(u,v)
                parent[v] ← u
```

➡

```
def prim(G, r):
    R ← V
    for each v in V:
        c[v] ← ∞
    c[r] ← 0

    while R is not empty:
        u ← delete-min(R, c)
        for each v in N_u:
            if v ∈ R and w(u,v) < c[v]:
                c[v] ← w(u,v)
                parent[v] ← u
```

Implementation Details (2)

❑ How to implement `delete-min()`?

- Use a priority queue Q based on min-heap where an item is a pair $(c[u],\ u)$.
  - Utilize cost $c[u]$ as key and node u as value

- Operations of min-heap for Prim's algorithm
  - `Q.insert(key, u)`: insert node u with key (=priority)
  - u ← `Q.remove()`: extract the node with minimum key

    ⇒ used instead of delete-min()
  - `Q.decrease-key(`u`, new-key)`: decrease node u's key to new-key

    ⇒ used after the relaxation

Implementation Details (3)

❑ Prim's algorithm using min-heap

- R can be replaced by the priority queue Q

```
def prim(G, r):
    R ← V
    for each v in V:
        c[v] ← ∞
    c[r] ← 0

    while R is not empty:
        u ← delete-min(R, c)
        for each v in N_u:
            if v ∈ R and w(u,v) < c[v]:
                c[v] ← w(u,v)
                parent[v] ← u
```

➡

```
def prim(G, r):
    Q ← min-heap()
    for each v in V − {r}:
        c[v] ← ∞ & Q.insert(c[v], v)
    c[r] ← 0 & Q.insert(c[r], r)

    while Q is not empty:
        u ← Q.remove()
        for each v in N_u:
            if v ∈ Q and w(u,v) < c[v]:
                c[v] ← w(u,v)
                Q.decrease-key(v, c[v])
                parent[v] ← u
```

See the version without
decrease-key() at Appendix

What You Need To Know

❑ Minimum spanning tree

- Spans the weighted and undirected graph as a tree with the minimum cost

❑ Kruskal’s algorithm (+ disjoint set)

- Incrementally grow MST by adding the minimum edge that does not produce a cycle

❑ Prim’s algorithm (+ binary heap)

- Incrementally grow MST by adding a new node connected with a minimum crossing edge

In Next Lecture

❑ Discussions on MST algorithms

- Time complexity analysis of Prim's algorithm
- Correctness analysis and other discussions

❑ Single source shortest path

- Dijkstra's algorithm

Thank You

Appendix: Implementation Details

❑ If you don't know how to implement decrease-key

- Just add a new item when the relaxation part.

```
def prim(G, r):
    Q ← min-heap()
    for each v in V:
        c[v] ← ∞ & Q.insert(c[v], v) & inMST[v] ← false
    c[r] ← 0 & Q.insert(c[r], r)

    while Q is not empty:
        u ← Q.remove()
        if inMST[u] is true : continue
        inMST[u] ← true
        for each v in N_u:
            if inMST[v] is false and w(u,v) < c[v]:
                c[v] ← w(u,v)
                Q.insert(c[v], v)
                parent[v] ← u
```

Appendix: Implementation Details

❑ Why does it work?

- According to the previous code, there will be multiple keys $c[v]$ on the same node v in Q.

- During the algorithm, the keys $c[v]$ were added while they monotonically decreases in the sequence.

- Thus, Q will always return the smallest $c[u]$ on node u whatever the costs of u are overlapped.

- If node u is already in the MST, the previously overlapped keys $c[v]$ are discarded.