Lecture 24.
Minimum Spanning Trees

Introduction to Algorithms
Da Nang University of Science and Technology

Dang Thien Binh
dtbinh@dut.udn.vn

Minimum Spanning Tree

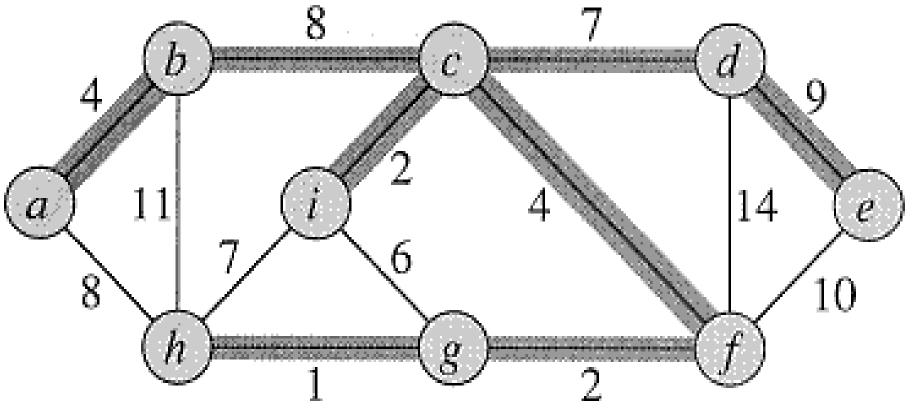
- A town has a set of houses and a set of roads
- A road connects 2 and only 2 houses
- A road connecting houses *u* and *v* has a repair cost *w(u, v)*
- ***Goal:*** Repair enough (and no more) roads such that
  - everyone stays connected: can reach every house from all other houses, and
  - total repair cost is minimum

Minimum Spanning Tree

- Model as a graph:
  - Undirected graph $G = (V, E)$
  - ***Weight*** $w(u, v)$ on each edge $(u, v) \in E$
  - Find $T \subseteq E$ such that
    - T connects all vertices
      - (T is a ***spanning tree***)
    - $w(T) = \sum_{(u,v)\in T} w(u,v)$ is minimized

Minimum Spanning Tree

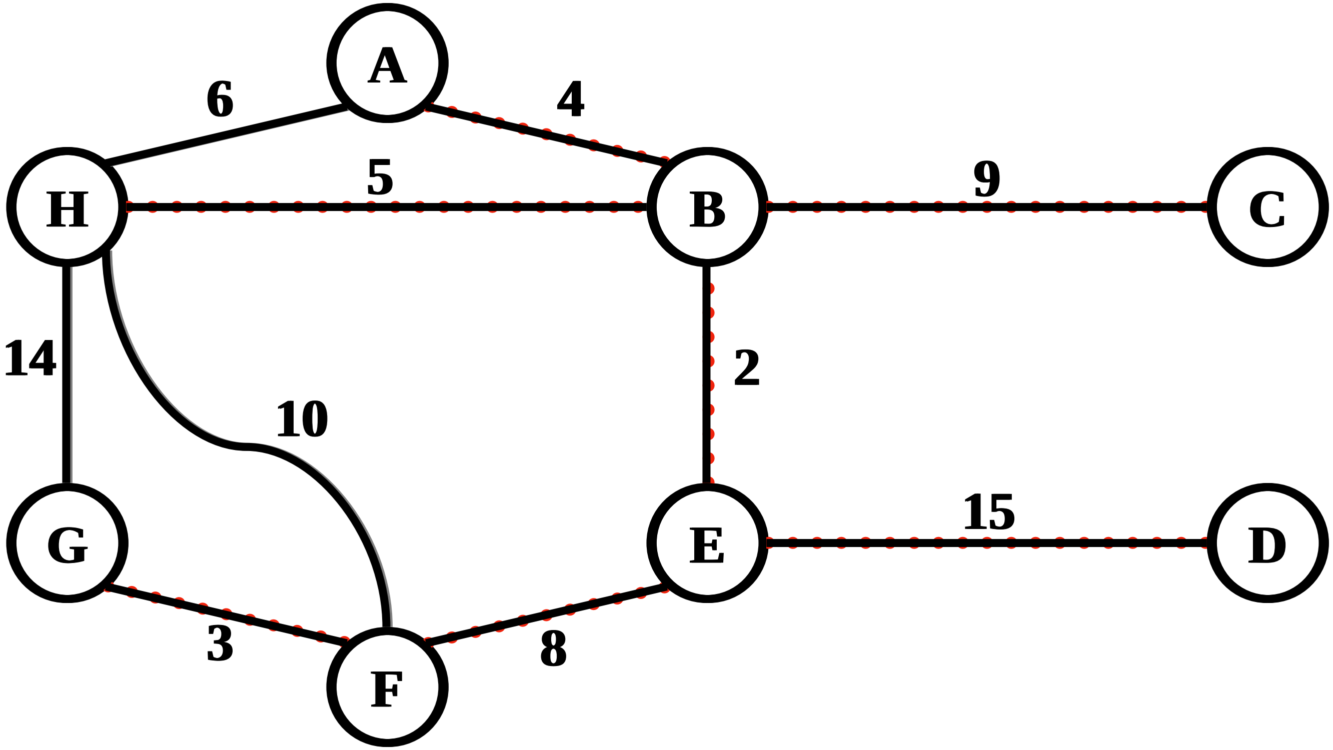
- A spanning tree whose weight is minimum over all spanning trees is called a ***Minimum Spanning Tree***, or ***MST***
- **Example:**



- In this example, there is *more than one* MST
- Replace edge (b,c) by (a,h)
- Get a different spanning tree with the same weight

Minimum Spanning Tree

- *Which edges form the Minimum Spanning Tree (MST) of the below graph?*



Minimum Spanning Tree

- MSTs satisfy the *optimal substructure* property: an optimal tree is composed of optimal subtrees
  - Let T be an MST of G with an edge (*u*,*v*) in the middle
  - Removing (*u*,*v*) partitions T into two trees T_1 and T_2
  - Claim: T_1 is an MST of $G_1 = (V_1,E_1)$, and T_2 is an MST of $G_2 = (V_2,E_2)$ (*Do V_1 and V_2 share vertices? Why?*)
  - Proof: $w(T) = w(u,v) + w(T_1) + w(T_2)$
    (There can't be a better tree than T_1 or T_2, or T would be suboptimal)

Growing An MST

- Some properties of an MST:
  - It has *|V| -1 edges*
  - It has *no cycles*
  - It might *not be unique*
- **Building up the solution**
  - We will build a set *A* of edges
  - Initially, *A* has no edges
  - As we add edges to *A*, maintain a loop invariant:
    - ***Loop invariant***: *A* is a subset of some MST
  - Add only edges that maintain the invariant

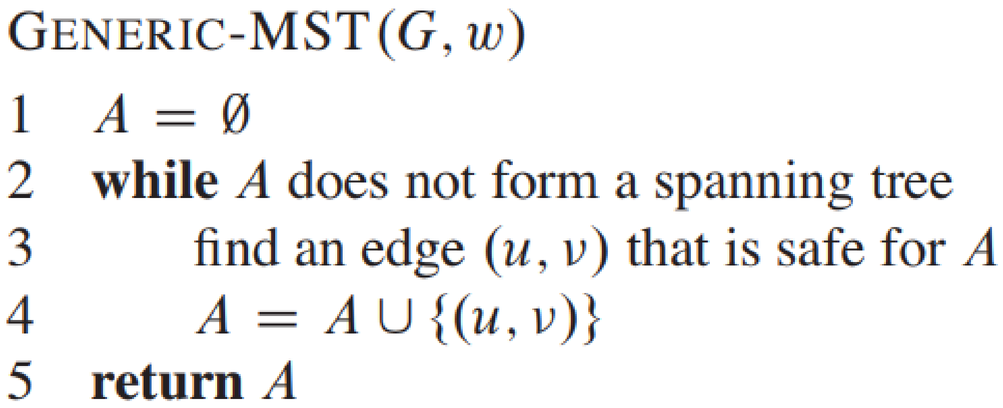
    If *A* is a subset of some MST, an edge *(u, v)* is ***safe*** for *A*

    if and only if $A \cup \{(u, v)\}$ is also a subset of some MST

    So we will add only safe edges

Growing An MST

- Generic MST algorithm



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

- Use the loop invariant to show that this generic algorithm works
  - **Initialization**
    - ★ The empty set trivially satisfies the loop invariant
  - **Maintenance**
    - ★ Since we add only safe edges, A remains a subset of some MST
  - **Termination**
    - ★ All edges added to A are in an MST, so when we stop, A is a spanning tree that is also an MST

Growing An MST

- Finding a safe edge
  - **Intuitiveness**:
    Let S ⊂ V be any set of vertices that includes h but not g (so that g is in V - S). In any MST, there has to be one edge (at least) that connects S with V - S. (Why not choose the edge with minimum weight?)
  - Some definitions: Let S ⊂ V and A ⊆ E
    - ★ A **cut** (S, V - S) is a partition of vertices into disjoint sets S and V - S
    - ★ Edge (u, v) ∈ E **crosses** cut (S, V - S)

      if one endpoint is in S and the other is in V - S
    - ★ A cut **respects** A if and only if no edge in A crosses the cut
    - ★ An edge is a **light edge** crossing a cut

      if and only if its weight is minimum over all edges crossing the cut

Growing An MST

- cut, crosses, respects, light edge

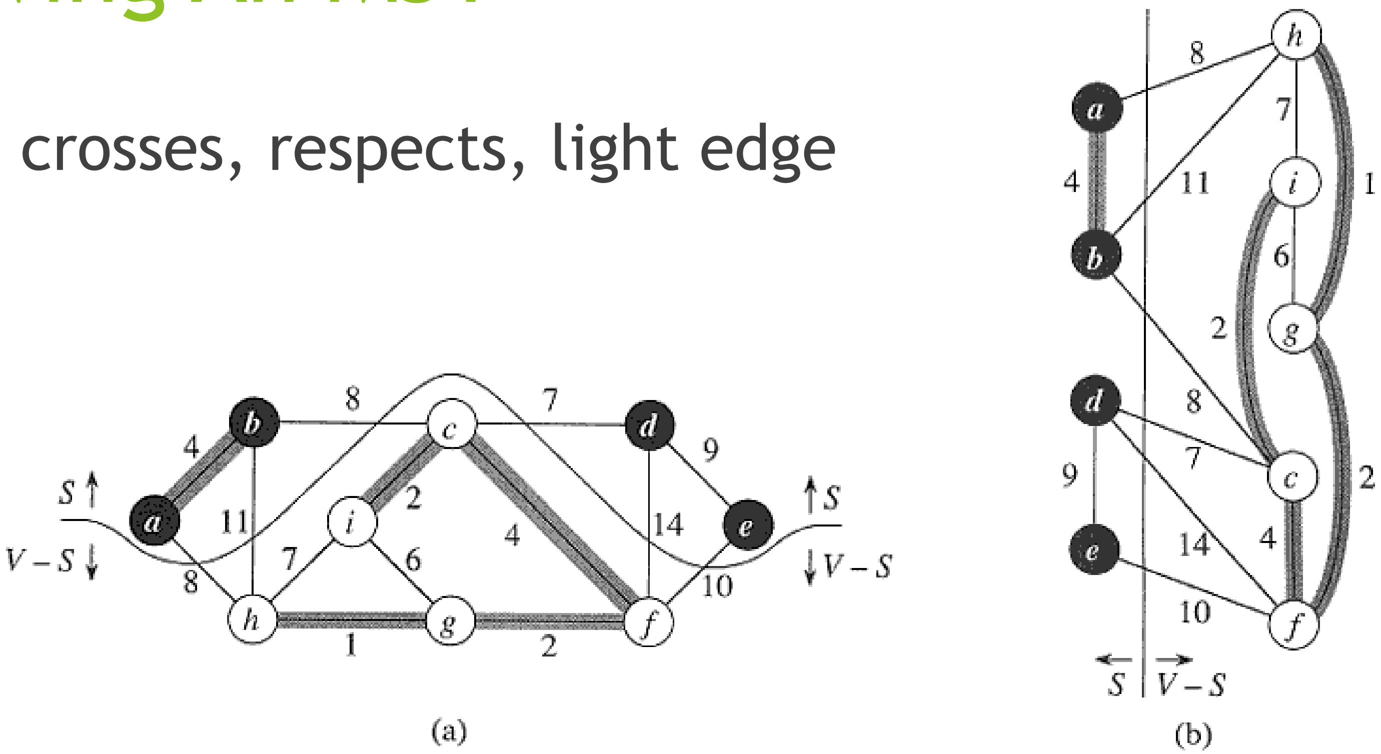


Figure 23.2 Two ways of viewing a cut $(S, V - S)$ of the graph from Figure 23.1. **(a)** The vertices in the set S are shown in black, and those in $V - S$ are shown in white. The edges crossing the cut are those connecting white vertices with black vertices. The edge (d, c) is the unique light edge crossing the cut. A subset A of the edges is shaded; note that the cut $(S, V - S)$ respects A, since no edge of A crosses the cut. **(b)** The same graph with the vertices in the set S on the left and the vertices in the set $V - S$ on the right. An edge crosses the cut if it connects a vertex on the left with a vertex on the right.

Growing An MST

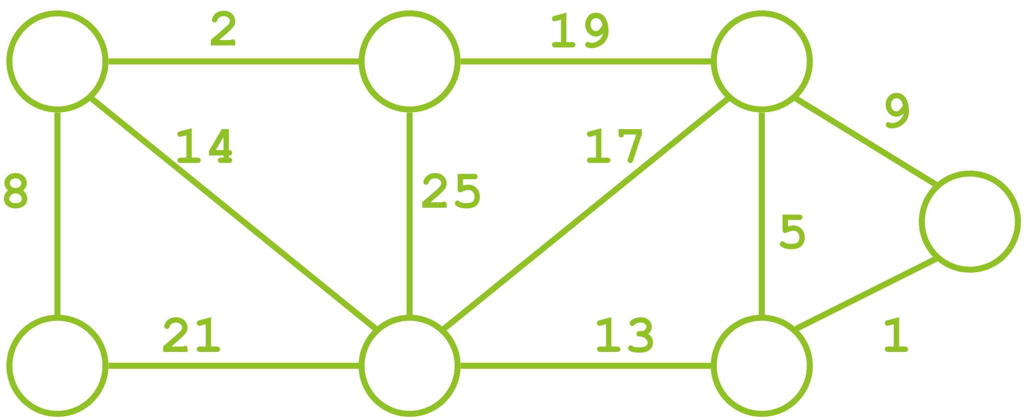
- Theorem
  - Let T be MST of G, and let A $\subseteq$ T be subtree of T
  - Let (u,v) be min-weight edge connecting A to V-A
  - Then $(u,v) \in$ T

Disjoint-Set Union Problem

- Want a data structure to support disjoint sets
  - Collection of disjoint sets $S = \{S_i\}$, $S_i \cap S_j = \varnothing$
- Need to support following operations:
  - MakeSet(x): $S = S \cup \{\{x\}\}$
  - Union(S_i, S_j): $S = S - \{S_i, S_j\} \cup \{S_i \cup S_j\}$
  - FindSet(x): return $S_i \in S$ such that $x \in S_i$
- Before discussing implementation details, we look at example application: MSTs

Kruskal's Algorithm

Run the algorithm:

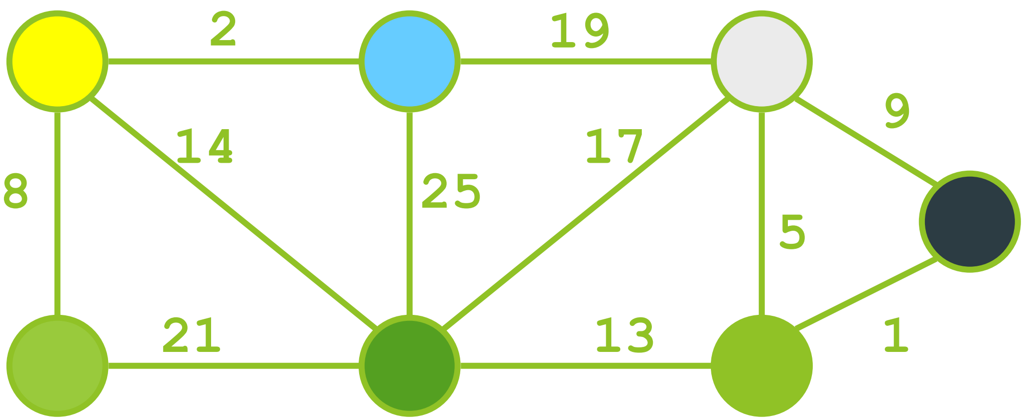


```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

Kruskal's Algorithm

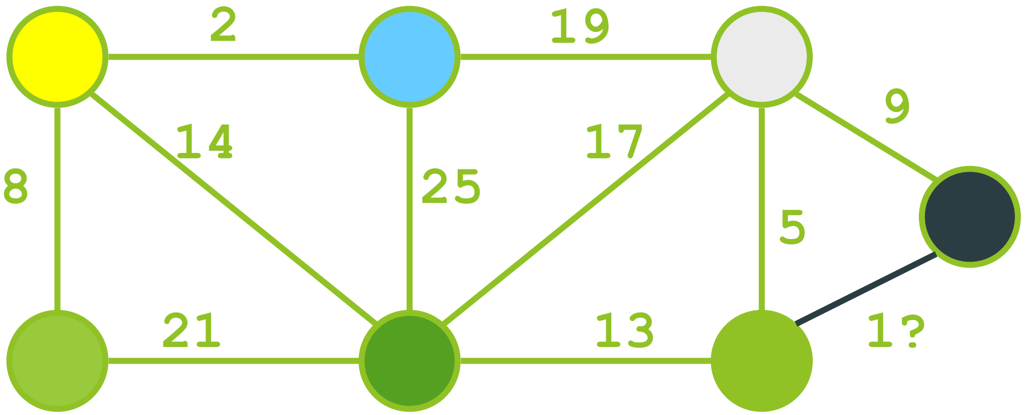
```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

Run the algorithm:



Kruskal's Algorithm

Run the algorithm:

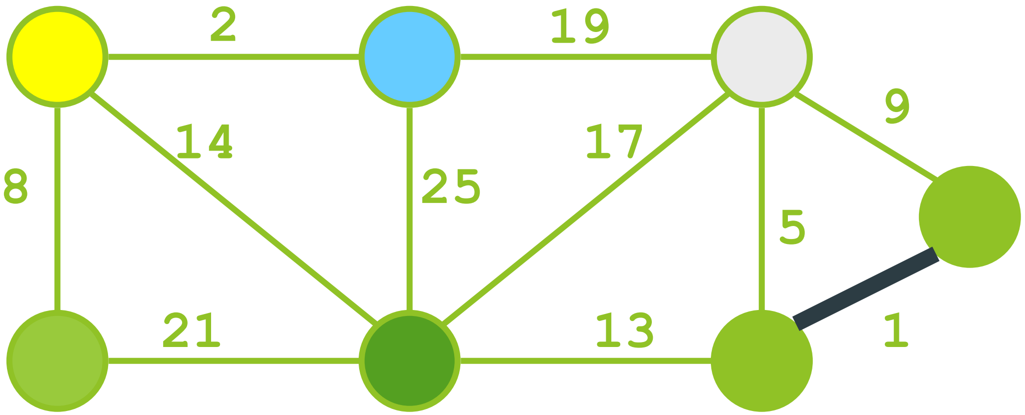


```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

Kruskal’s Algorithm

```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

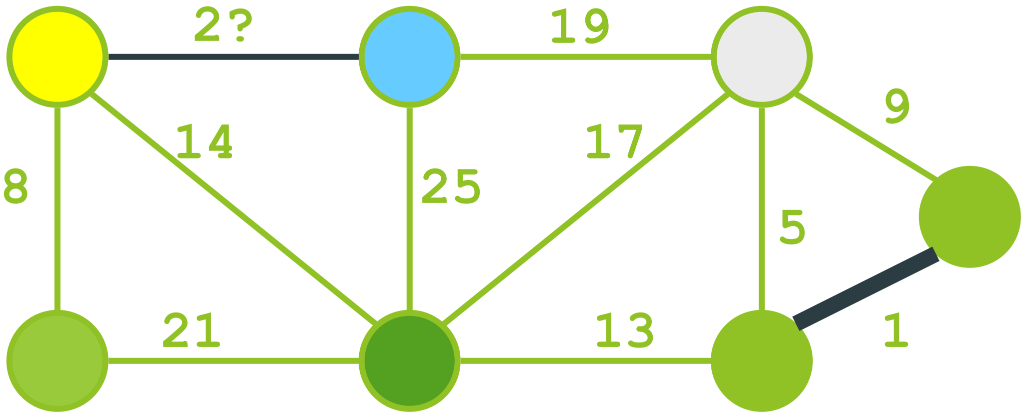
Run the algorithm:



Kruskal’s Algorithm

```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

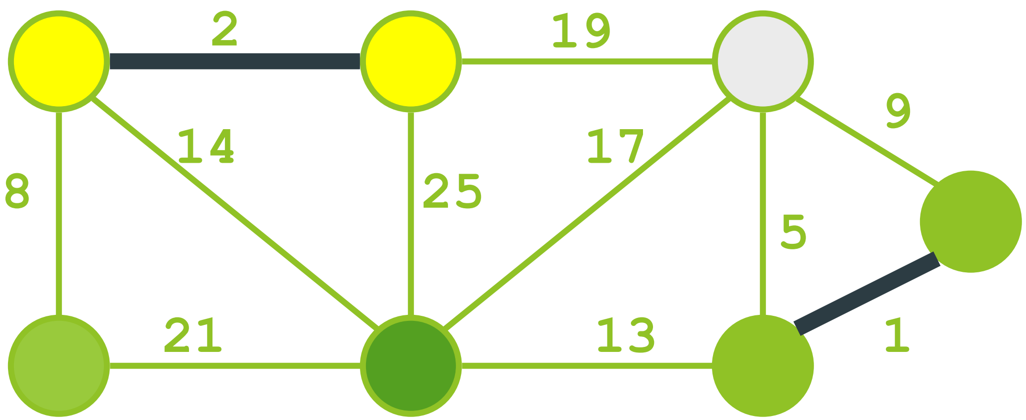
Run the algorithm:



Kruskal's Algorithm

```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

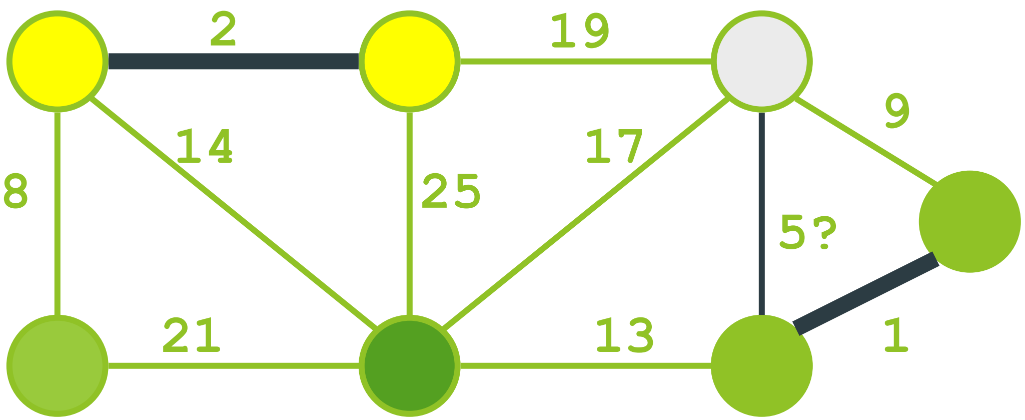
Run the algorithm:



Kruskal’s Algorithm

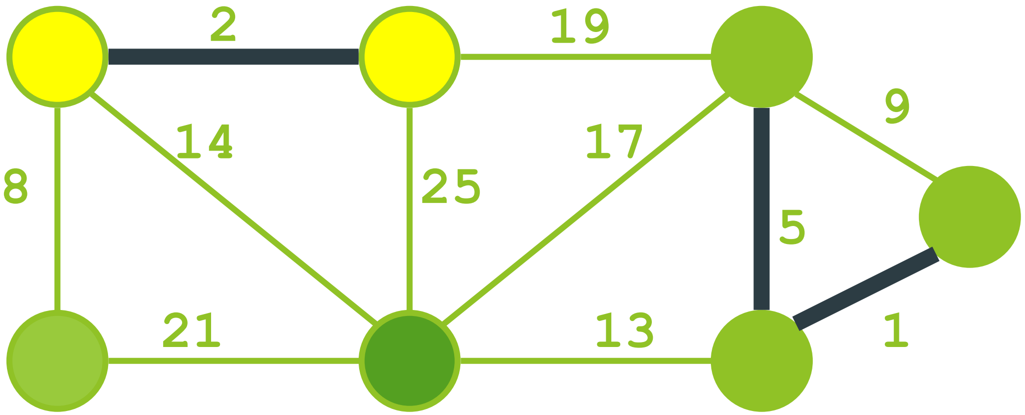
```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

Run the algorithm:



Kruskal’s Algorithm

Run the algorithm:

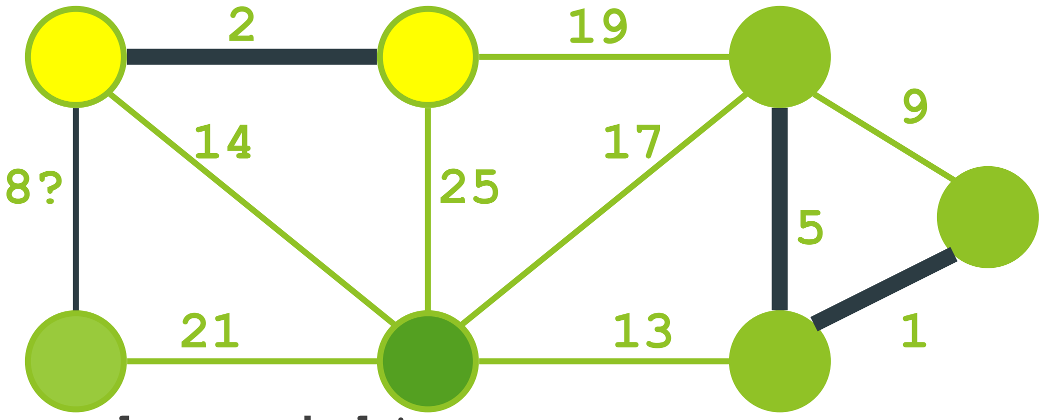


```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

Kruskal’s Algorithm

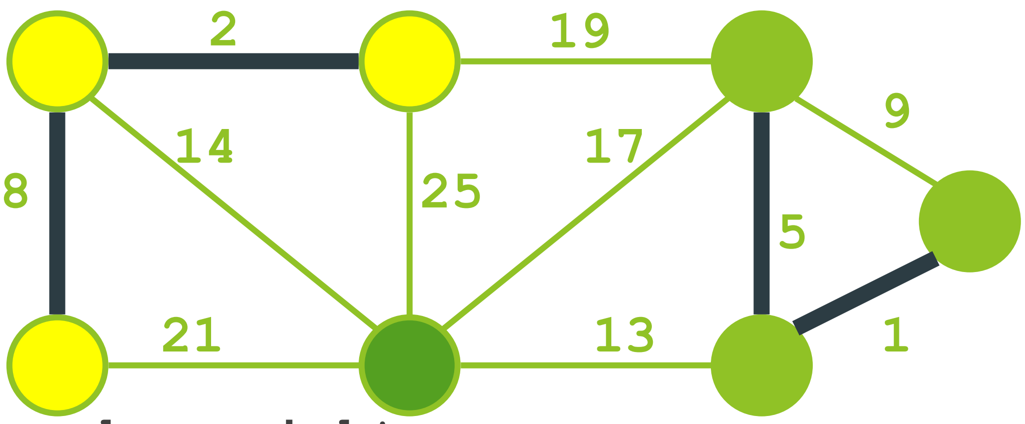
```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

Run the algorithm:



Kruskal’s Algorithm

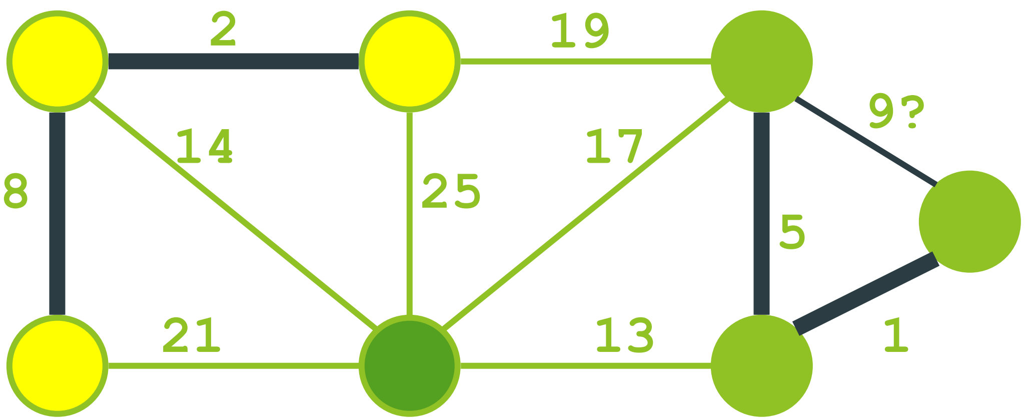
Run the algorithm:



```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

Kruskal's Algorithm

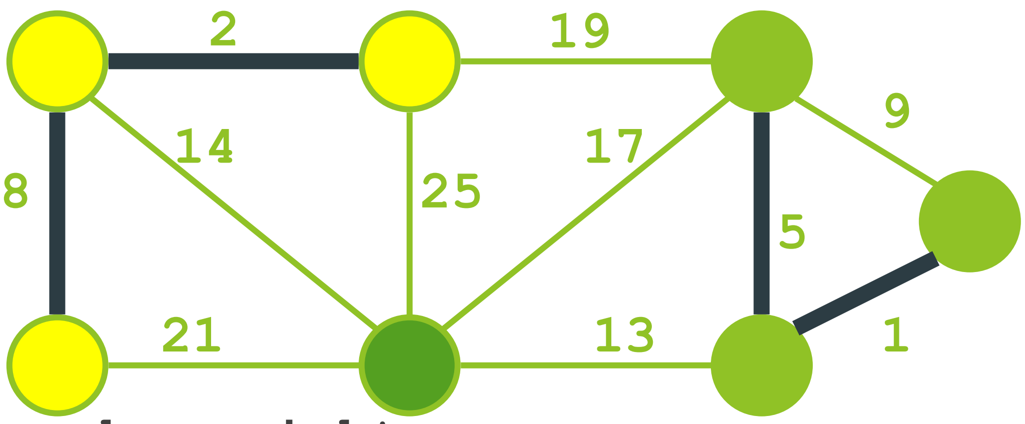
Run the algorithm:



```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

Kruskal’s Algorithm

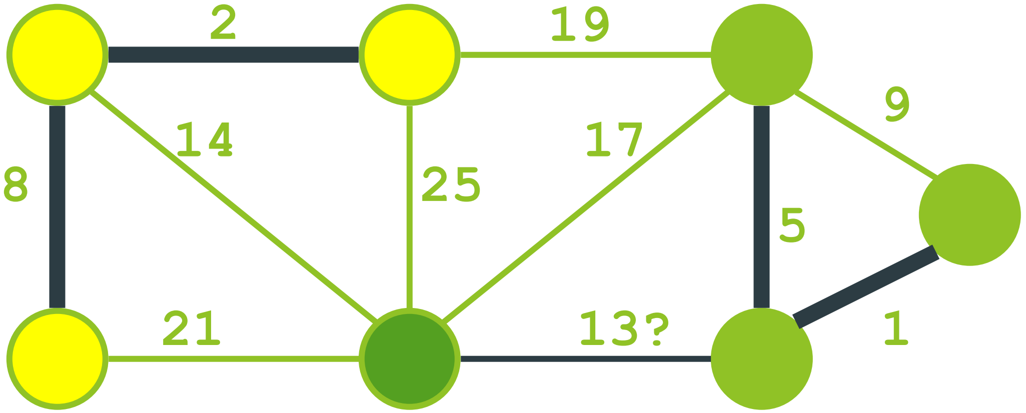
Run the algorithm:



```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

Kruskal's Algorithm

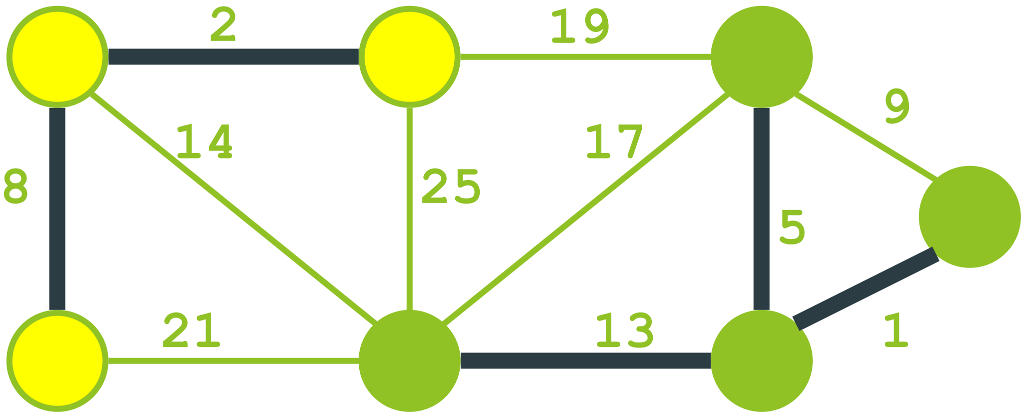
Run the algorithm:



```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

Kruskal’s Algorithm

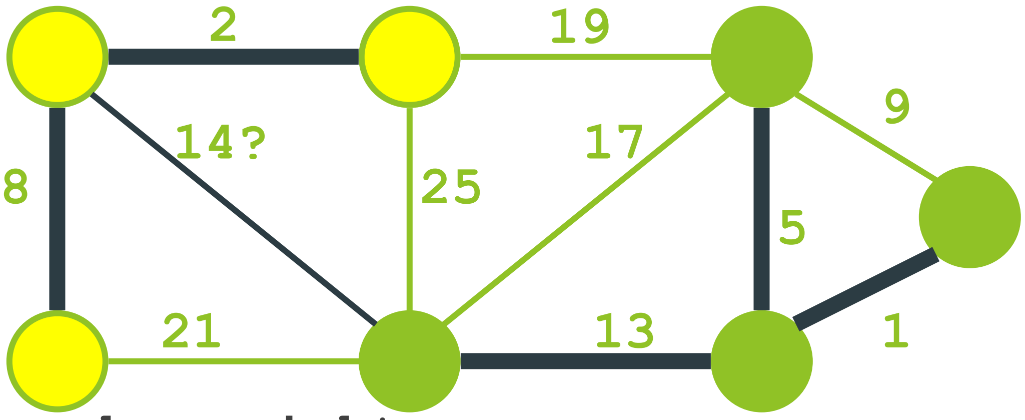
Run the algorithm:



```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

Kruskal’s Algorithm

Run the algorithm:

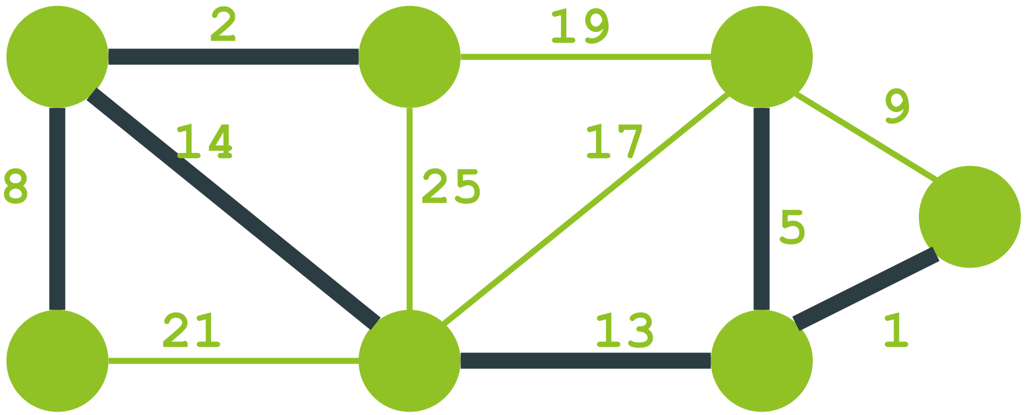


```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

Kruskal’s Algorithm

```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

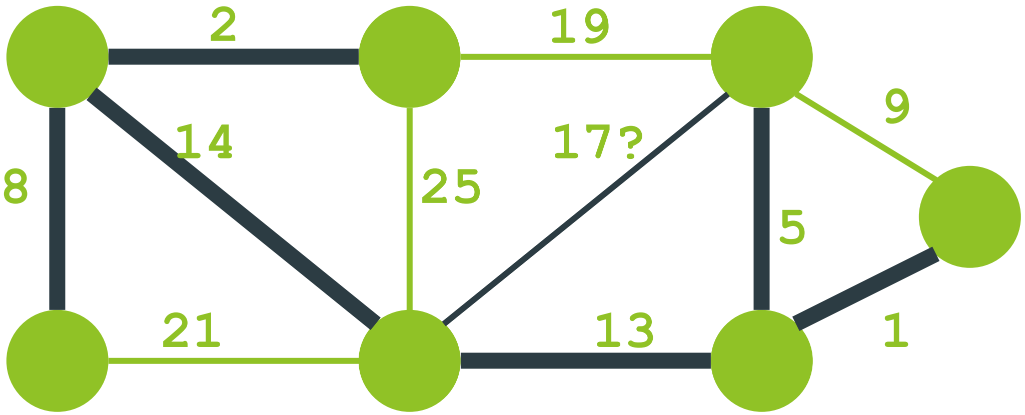
Run the algorithm:



Kruskal’s Algorithm

```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

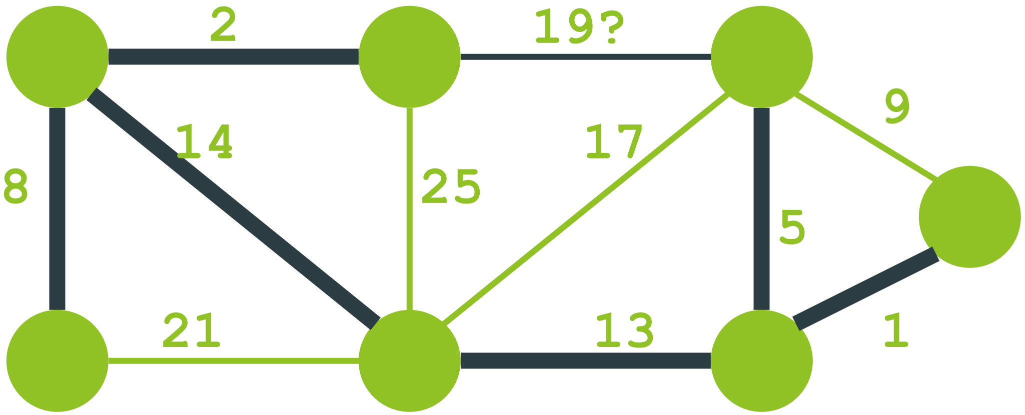
Run the algorithm:



Kruskal’s Algorithm

```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

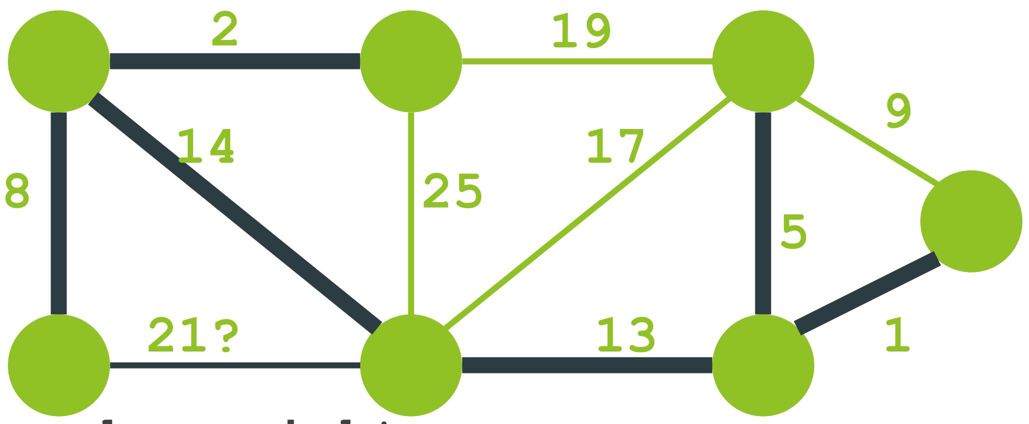
Run the algorithm:



Kruskal's Algorithm

```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

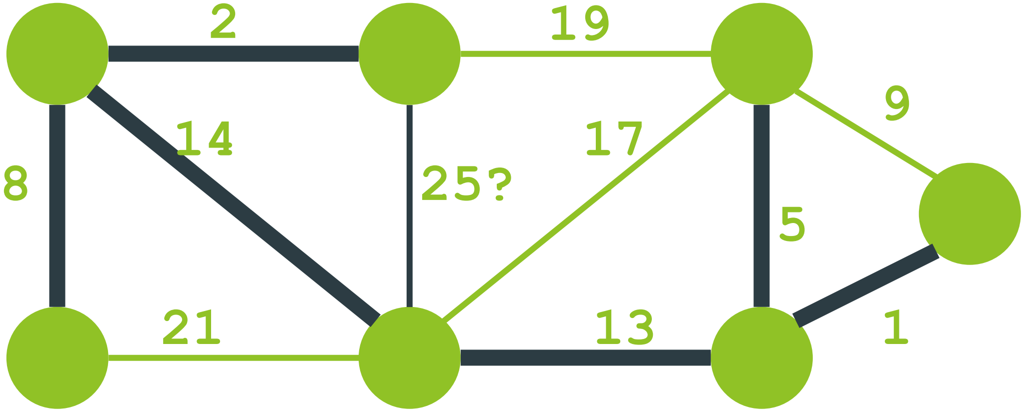
Run the algorithm:



Kruskal’s Algorithm

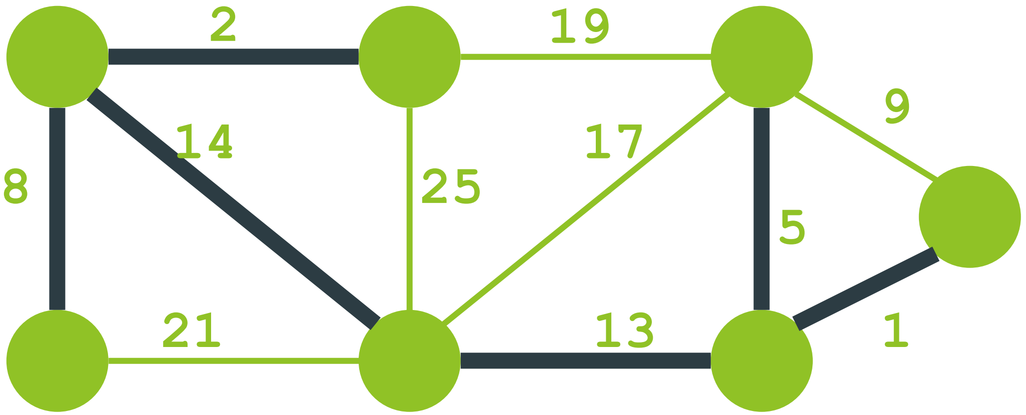
```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

Run the algorithm:



Kruskal’s Algorithm

Run the algorithm:

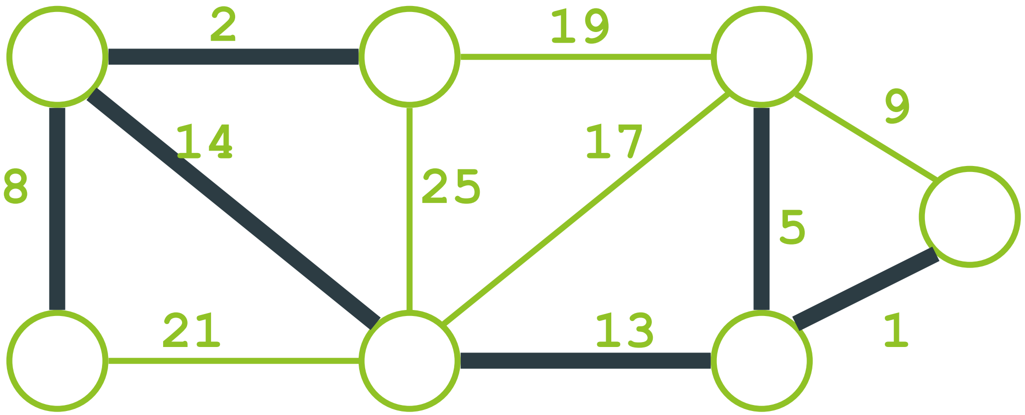


```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

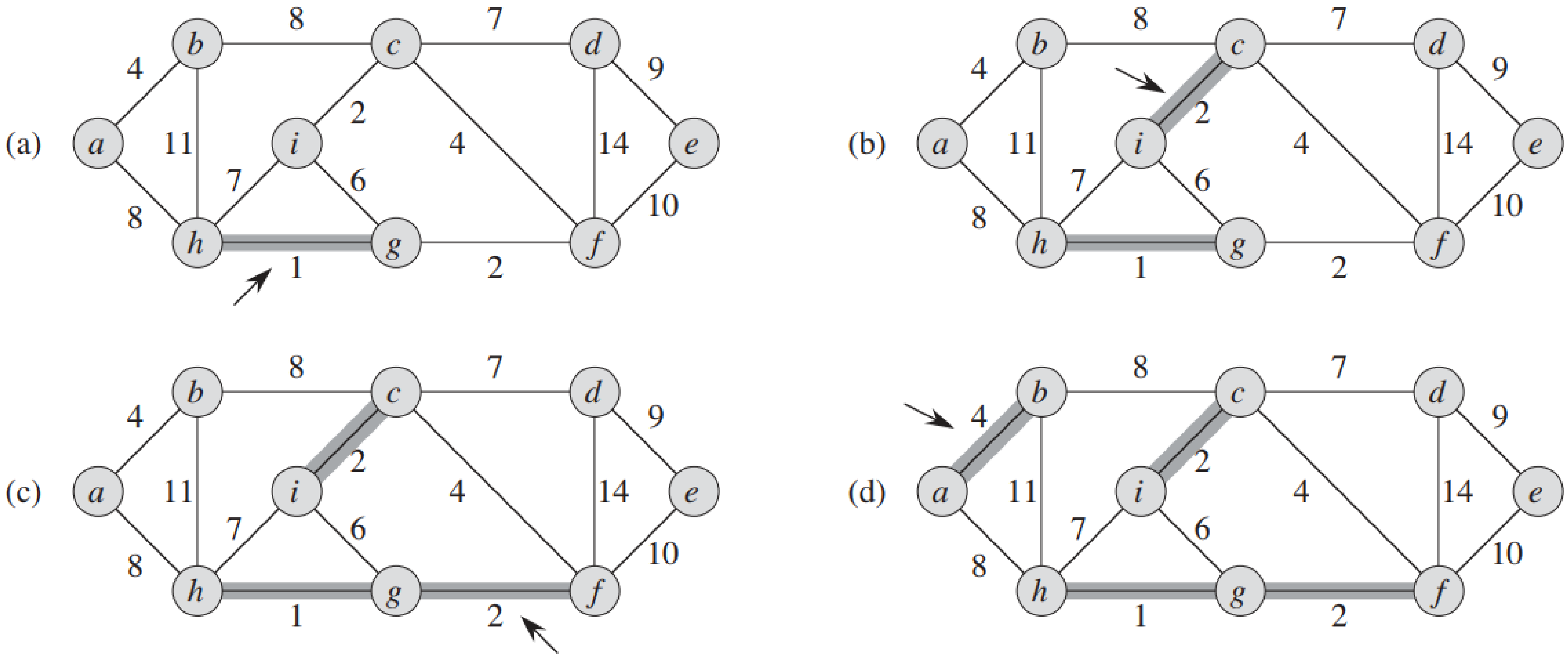
Kruskal's Algorithm

```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

Run the algorithm:



Kruskal’s Algorithm



Kruskal’s Algorithm

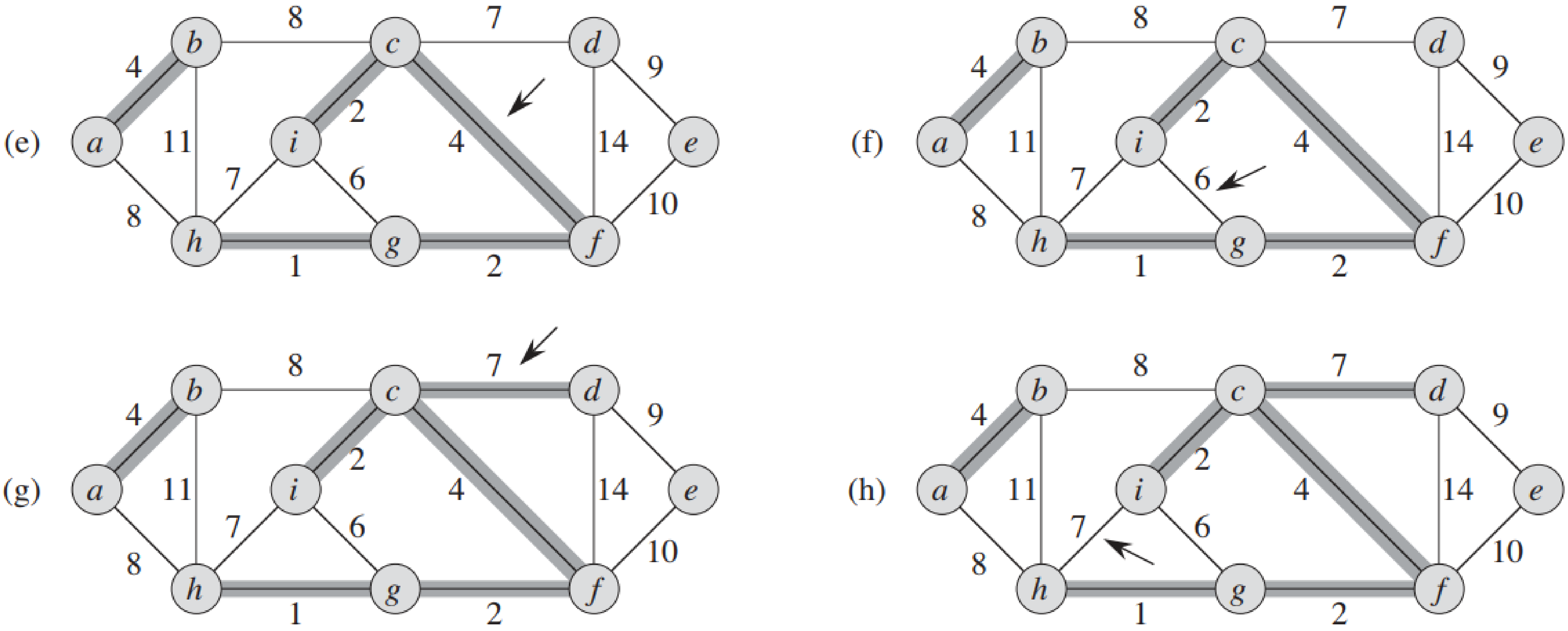
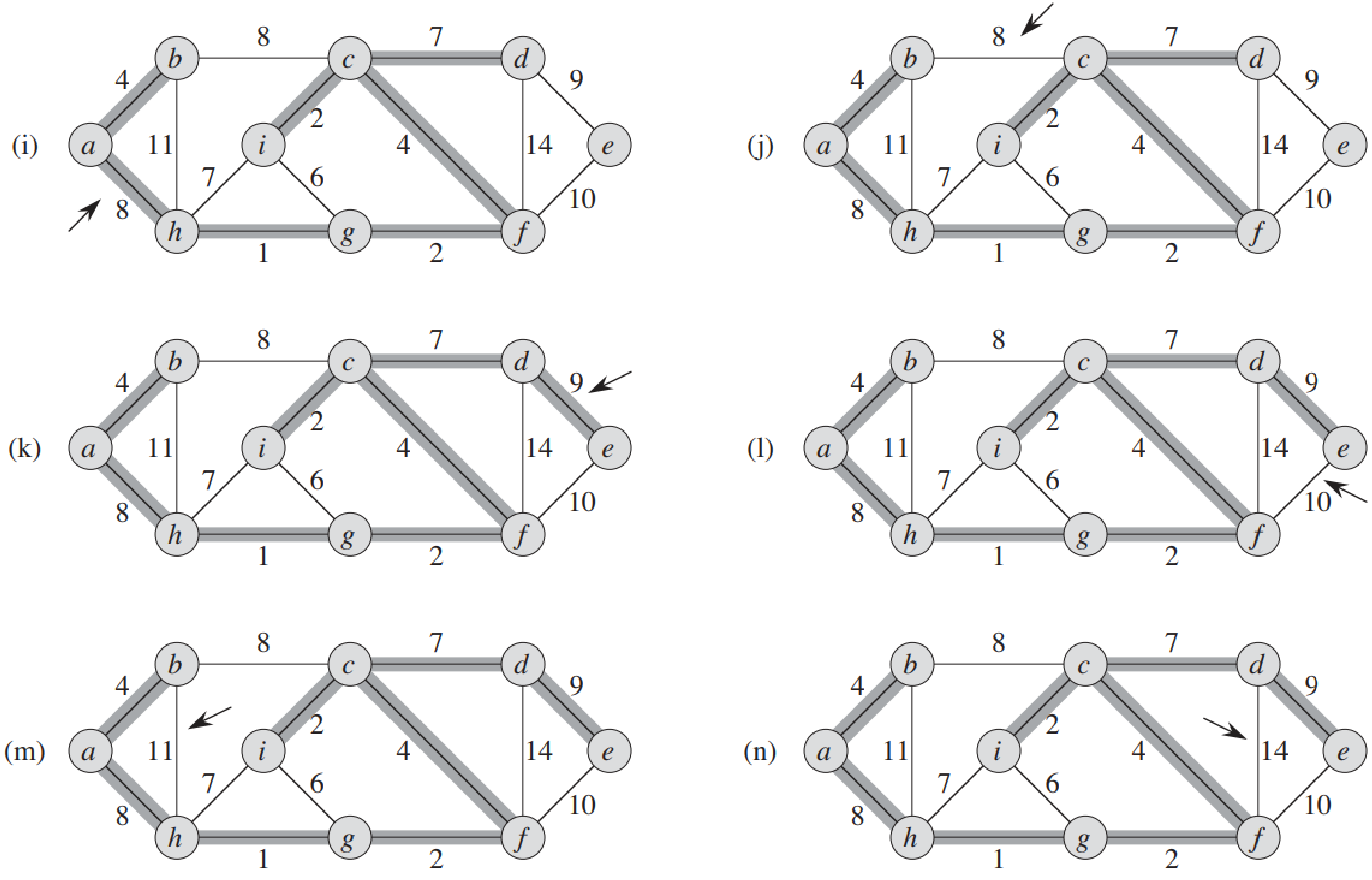


Figure 23.4 The execution of Kruskal’s algorithm on the graph from Figure 23.1. Shaded edges belong to the forest *A* being grown. The algorithm considers each edge in sorted order by weight. An arrow points to the edge under consideration at each step of the algorithm. If the edge joins two distinct trees in the forest, it is added to the forest, thereby merging the two trees.

Kruskal’s Algorithm



Kruskal’s Algorithm

```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

What will affect the running time?

1 Sort

O(V) MakeSet() calls

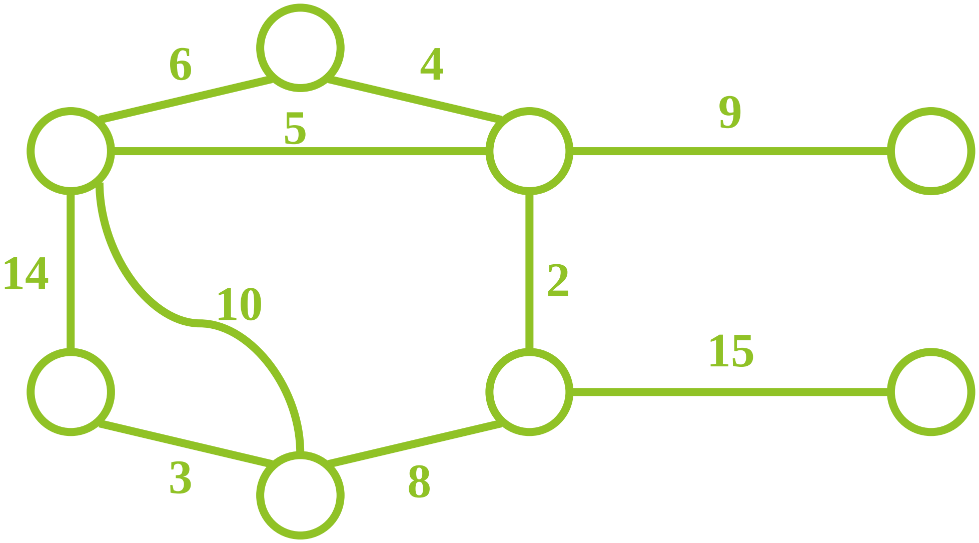
O(E) FindSet() calls

O(V) Union() calls

(*Exactly how many Union()s?*)

Prim’s Algorithm

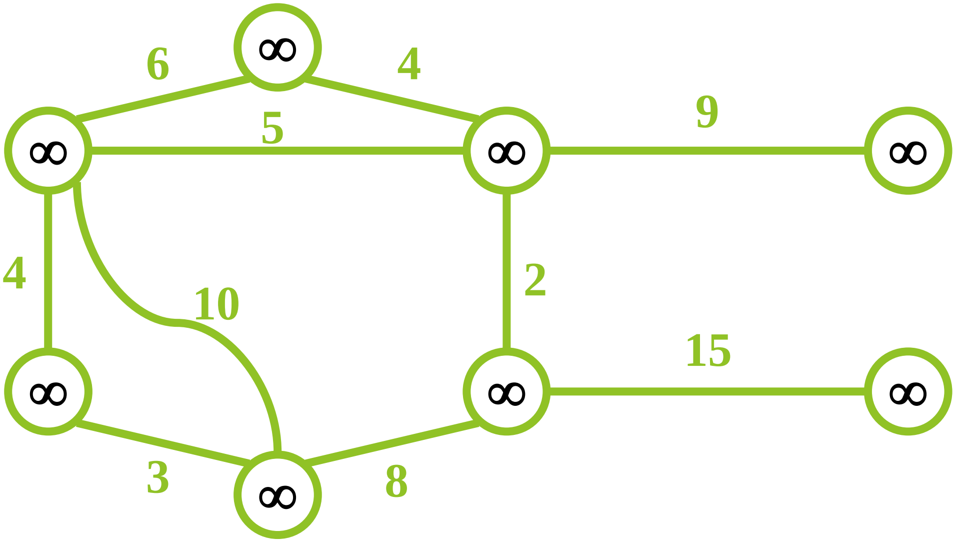
```
MST-Prim(G, w, r)
    Q = G.V;
    for each u ∈ Q
        u.key = ∞;
    r.key = 0;
    r.p = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj(u)
            if (v ∈ Q and w(u,v) < v.key)
                v.p = u;
                v.key = w(u,v);
```



Run on example graph

Prim's Algorithm

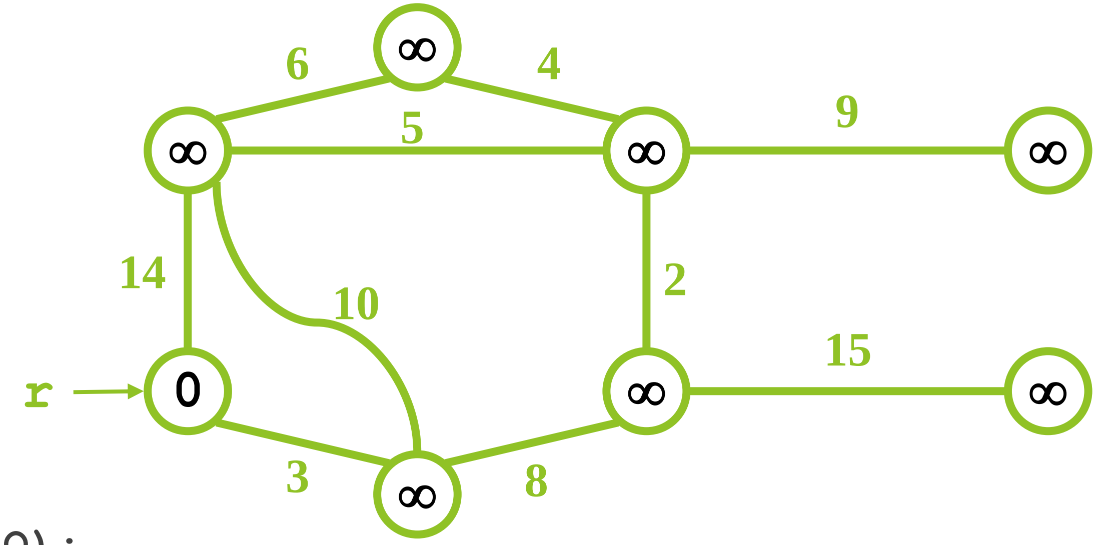
```
MST-Prim(G, w, r)
    Q = G.V;
    for each u ∈ Q
        u.key = ∞;
    r.key = 0;
    r.p = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj(u)
            if (v ∈ Q and w(u,v) < v.key)
                v.p = u;
                v.key = w(u,v);
```



Run on example graph

Prim’s Algorithm

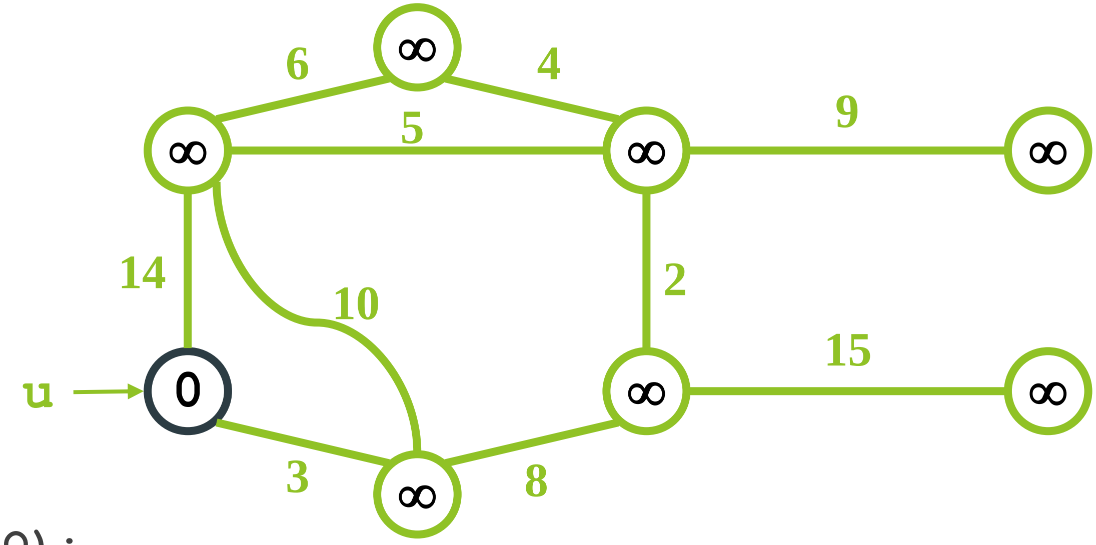
```
MST-Prim(G, w, r)
    Q = G.V;
    for each u ∈ Q
        u.key = ∞;
    r.key = 0;
    r.p = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj(u)
            if (v ∈ Q and w(u,v) < v.key)
                v.p = u;
                v.key = w(u,v);
```



Pick a start vertex r

Prim’s Algorithm

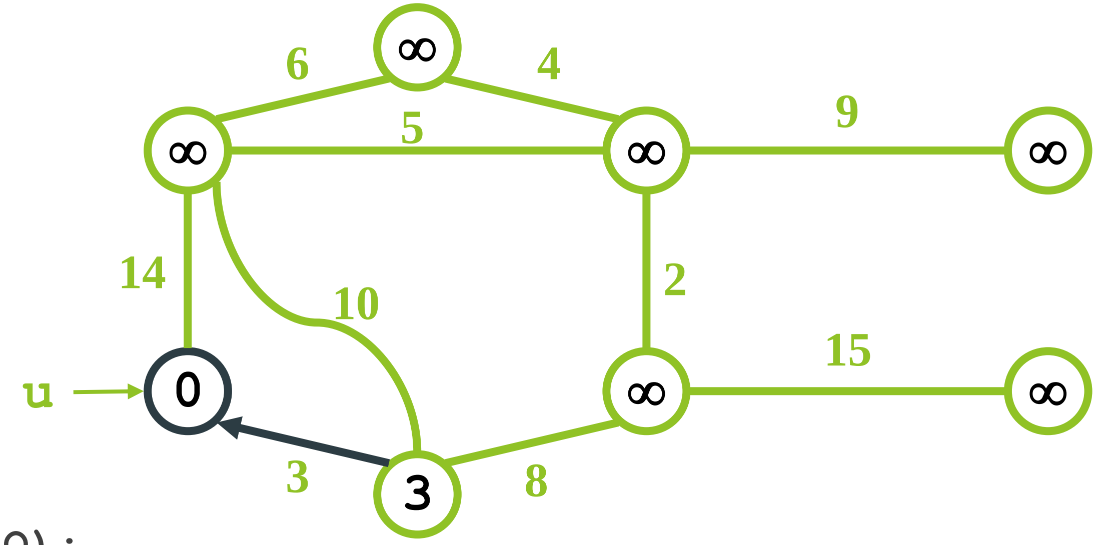
```
MST-Prim(G, w, r)
    Q = G.V;
    for each u ∈ Q
        u.key = ∞;
    r.key = 0;
    r.p = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj(u)
            if (v ∈ Q and w(u,v) < v.key)
                v.p = u;
                v.key = w(u,v);
```



Black vertices have been removed from Q

Prim's Algorithm

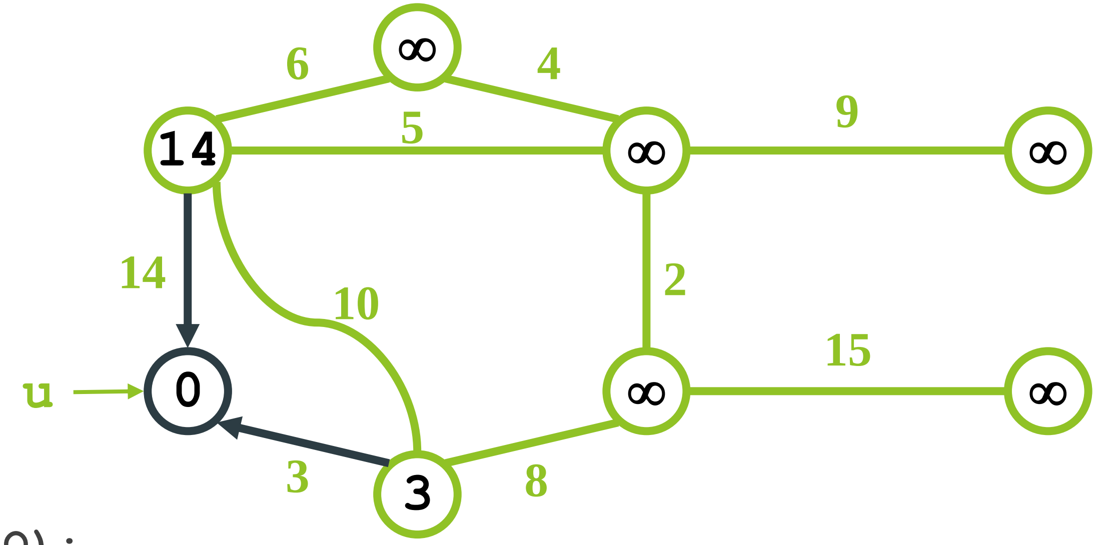
```
MST-Prim(G, w, r)
    Q = G.V;
    for each u ∈ Q
        u.key = ∞;
    r.key = 0;
    r.p = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj(u)
            if (v ∈ Q and w(u,v) < v.key)
                v.p = u;
                v.key = w(u,v);
```



Blue arrows indicate parent pointers

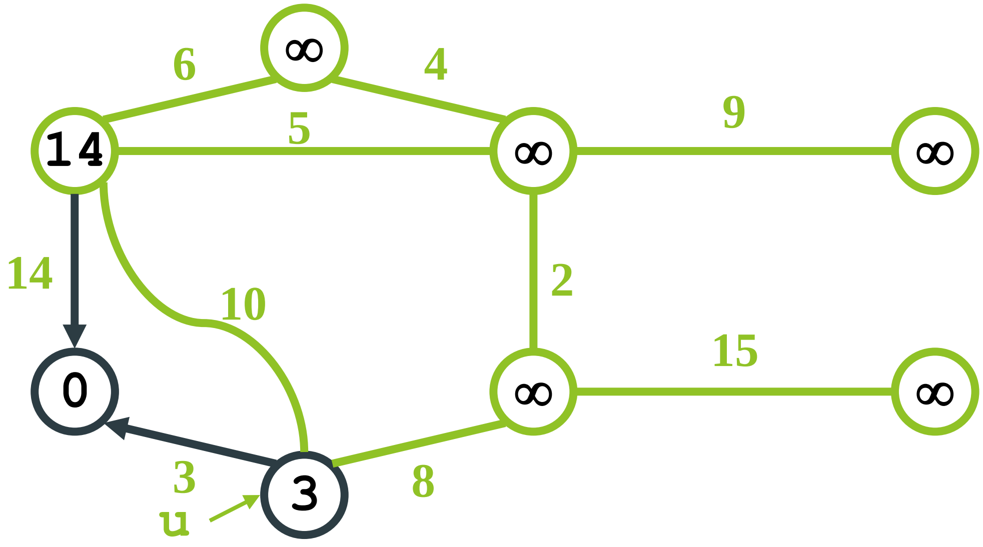
Prim’s Algorithm

```
MST-Prim(G, w, r)
    Q = G.V;
    for each u ∈ Q
        u.key = ∞;
    r.key = 0;
    r.p = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj(u)
            if (v ∈ Q and w(u,v) < v.key)
                v.p = u;
                v.key = w(u,v);
```



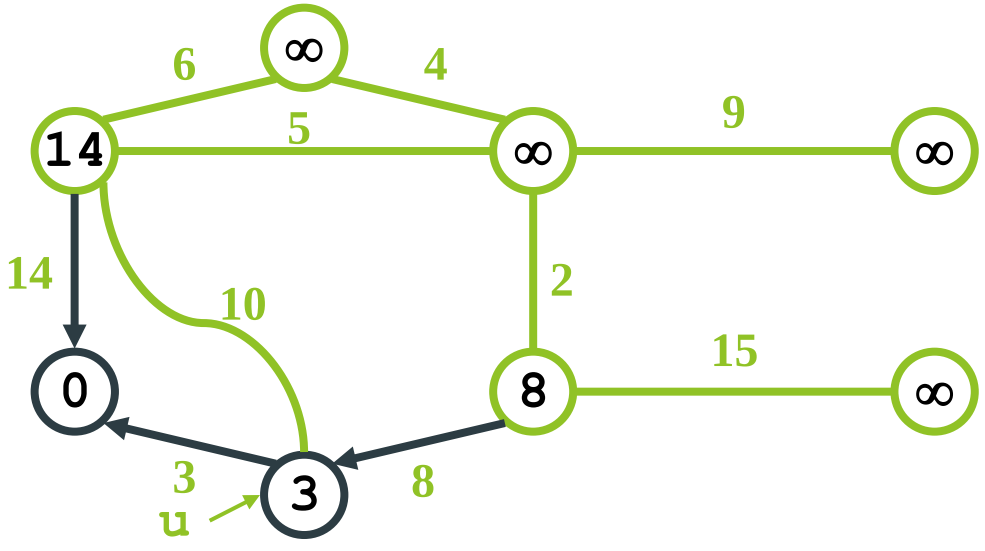
Prim’s Algorithm

```
MST-Prim(G, w, r)
    Q = G.V;
    for each u ∈ Q
        u.key = ∞;
    r.key = 0;
    r.p = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj(u)
            if (v ∈ Q and w(u,v) < v.key)
                v.p = u;
                v.key = w(u,v);
```



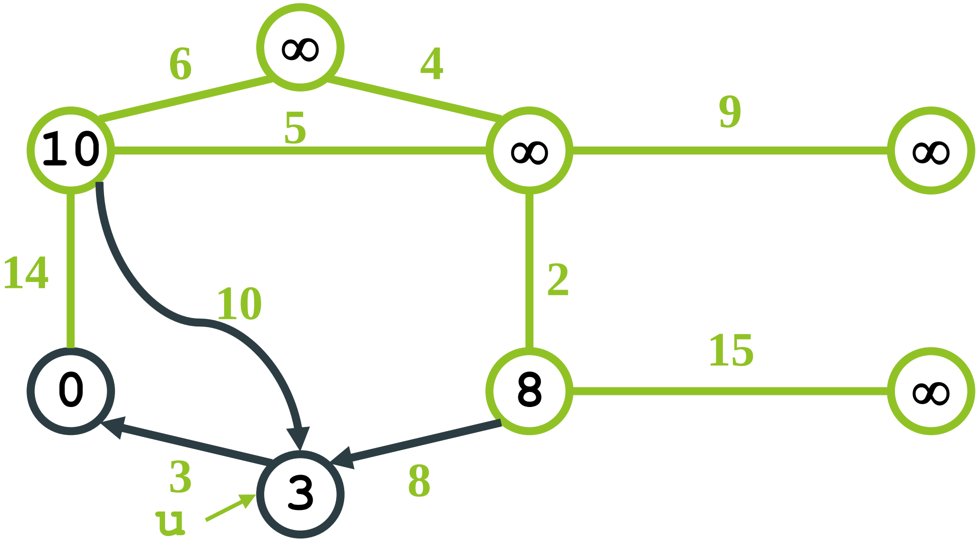
Prim’s Algorithm

```
MST-Prim(G, w, r)
    Q = G.V;
    for each u ∈ Q
        u.key = ∞;
    r.key = 0;
    r.p = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj(u)
            if (v ∈ Q and w(u,v) < v.key)
                v.p = u;
                v.key = w(u,v);
```



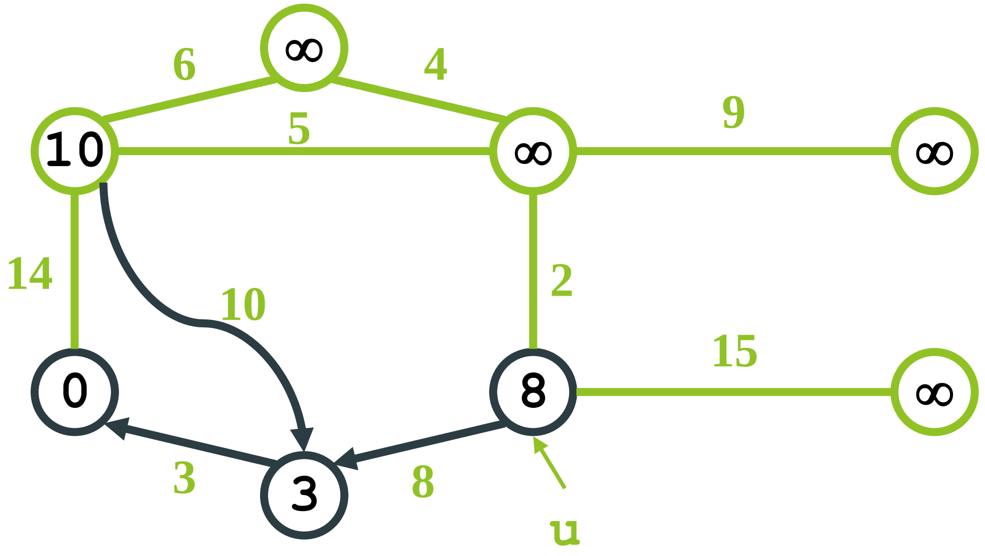
Prim's Algorithm

```
MST-Prim(G, w, r)
    Q = G.V;
    for each u ∈ Q
        u.key = ∞;
    r.key = 0;
    r.p = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj(u)
            if (v ∈ Q and w(u,v) < v.key)
                v.p = u;
                v.key = w(u,v);
```



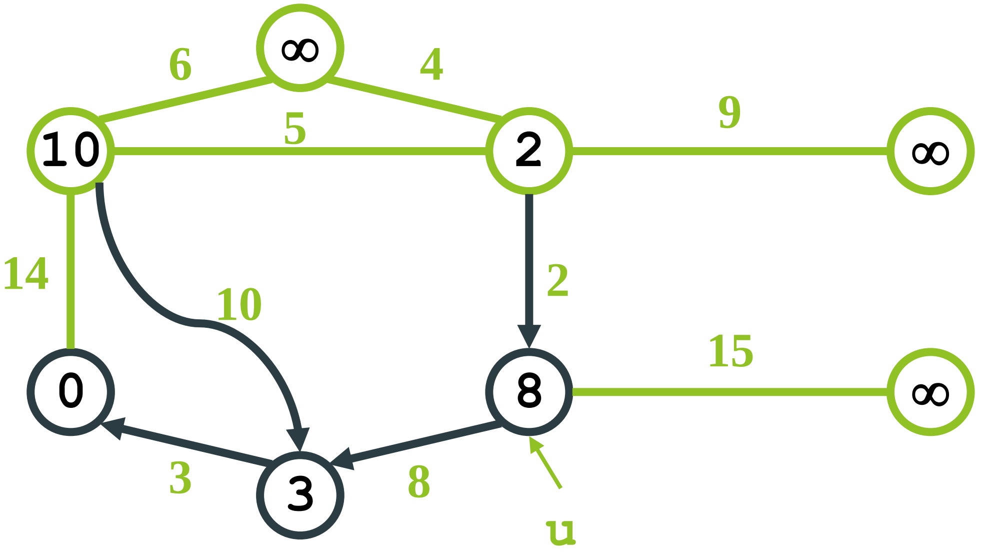
Prim’s Algorithm

```
MST-Prim(G, w, r)
    Q = G.V;
    for each u ∈ Q
        u.key = ∞;
    r.key = 0;
    r.p = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj(u)
            if (v ∈ Q and w(u,v) < v.key)
                v.p = u;
                v.key = w(u,v);
```



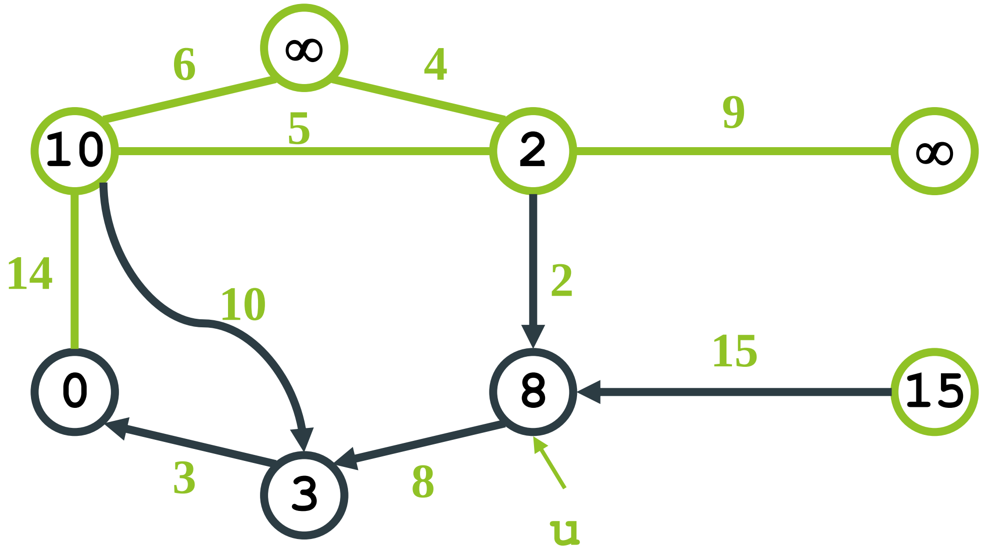
Prim's Algorithm

```
MST-Prim(G, w, r)
    Q = G.V;
    for each u ∈ Q
        u.key = ∞;
    r.key = 0;
    r.p = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj(u)
            if (v ∈ Q and w(u,v) < v.key)
                v.p = u;
                v.key = w(u,v);
```



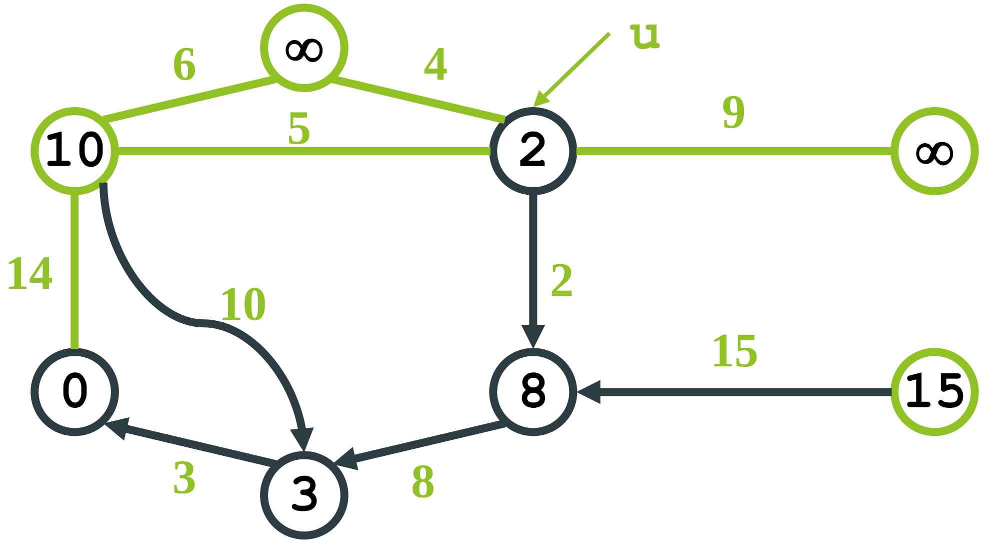
Prim’s Algorithm

```
MST-Prim(G, w, r)
    Q = G.V;
    for each u ∈ Q
        u.key = ∞;
    r.key = 0;
    r.p = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj(u)
            if (v ∈ Q and w(u,v) < v.key)
                v.p = u;
                v.key = w(u,v);
```



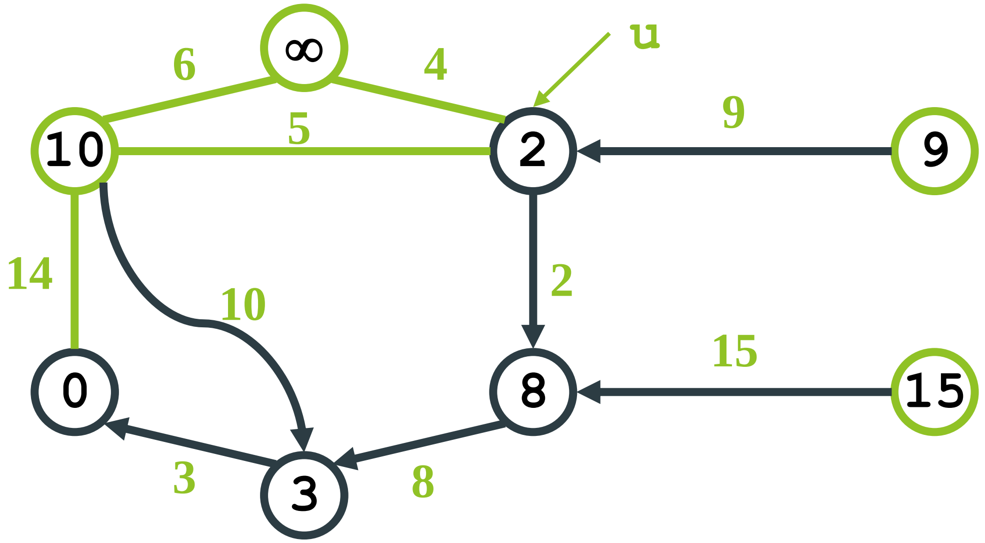
Prim’s Algorithm

```
MST-Prim(G, w, r)
    Q = G.V;
    for each u ∈ Q
        u.key = ∞;
    r.key = 0;
    r.p = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj(u)
            if (v ∈ Q and w(u,v) < v.key)
                v.p = u;
                v.key = w(u,v);
```



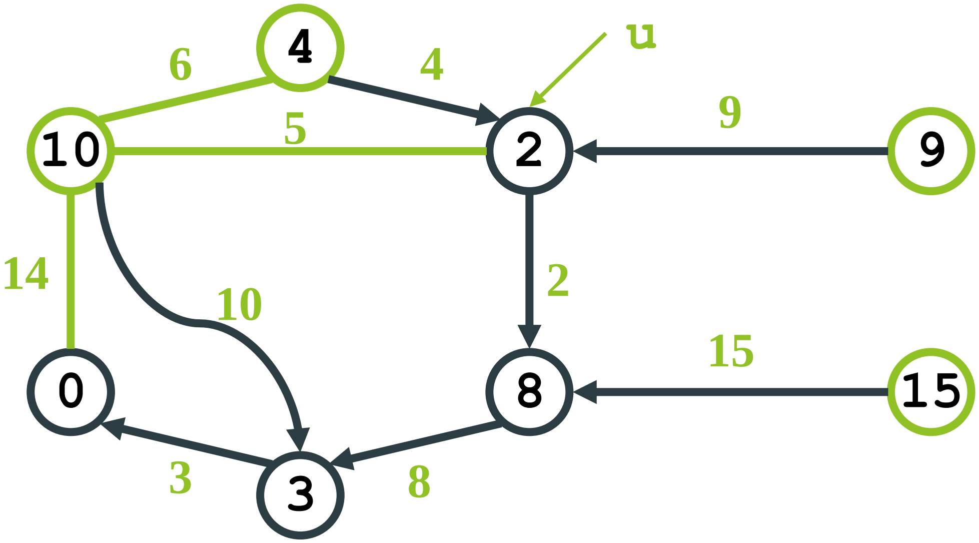
Prim’s Algorithm

```
MST-Prim(G, w, r)
    Q = G.V;
    for each u ∈ Q
        u.key = ∞;
    r.key = 0;
    r.p = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj(u)
            if (v ∈ Q and w(u,v) < v.key)
                v.p = u;
                v.key = w(u,v);
```



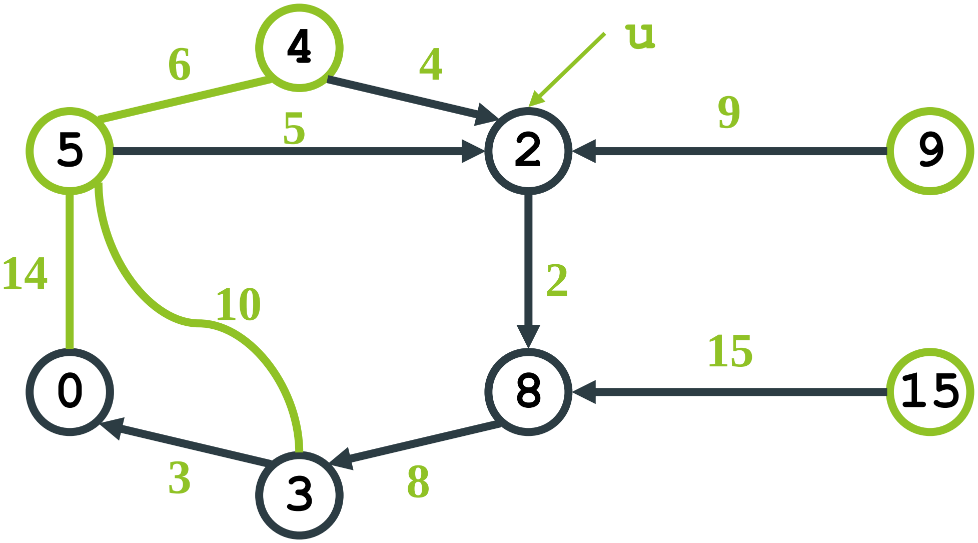
Prim’s Algorithm

```
MST-Prim(G, w, r)
    Q = G.V;
    for each u ∈ Q
        u.key = ∞;
    r.key = 0;
    r.p = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj(u)
            if (v ∈ Q and w(u,v) < v.key)
                v.p = u;
                v.key = w(u,v);
```



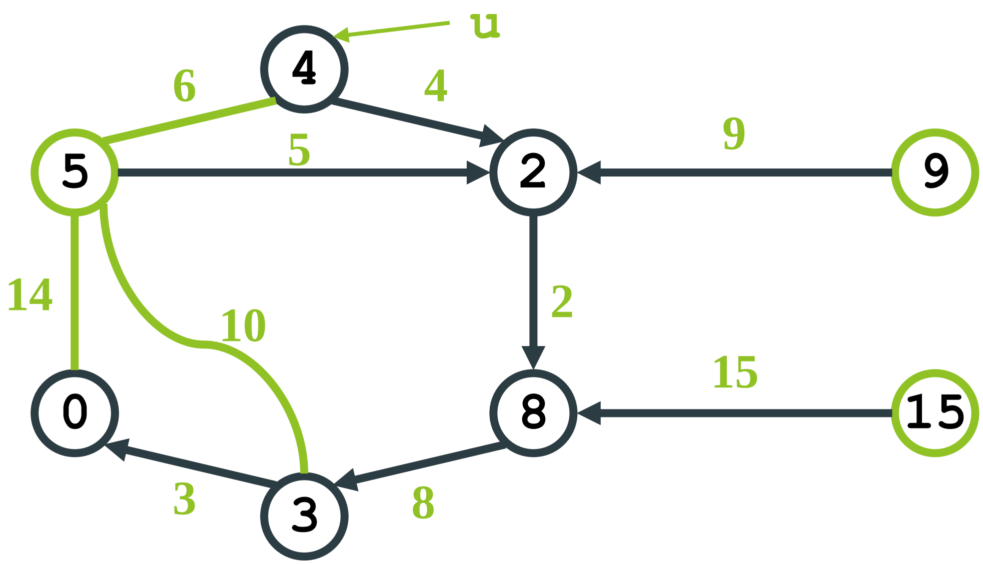
Prim's Algorithm

```
MST-Prim(G, w, r)
    Q = G.V;
    for each u ∈ Q
        u.key = ∞;
    r.key = 0;
    r.p = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj(u)
            if (v ∈ Q and w(u,v) < v.key)
                v.p = u;
                v.key = w(u,v);
```



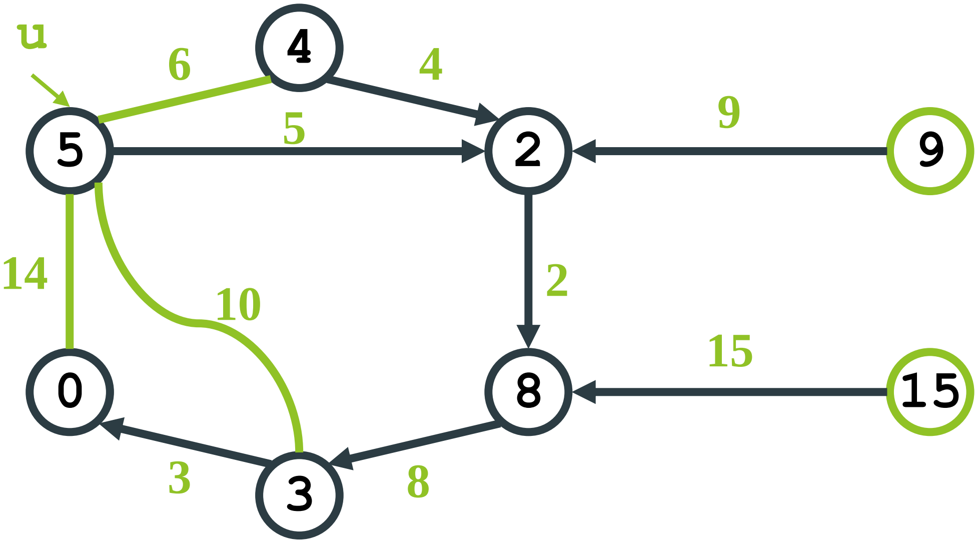
Prim’s Algorithm

```
MST-Prim(G, w, r)
    Q = G.V;
    for each u ∈ Q
        u.key = ∞;
    r.key = 0;
    r.p = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj(u)
            if (v ∈ Q and w(u,v) < v.key)
                v.p = u;
                v.key = w(u,v);
```



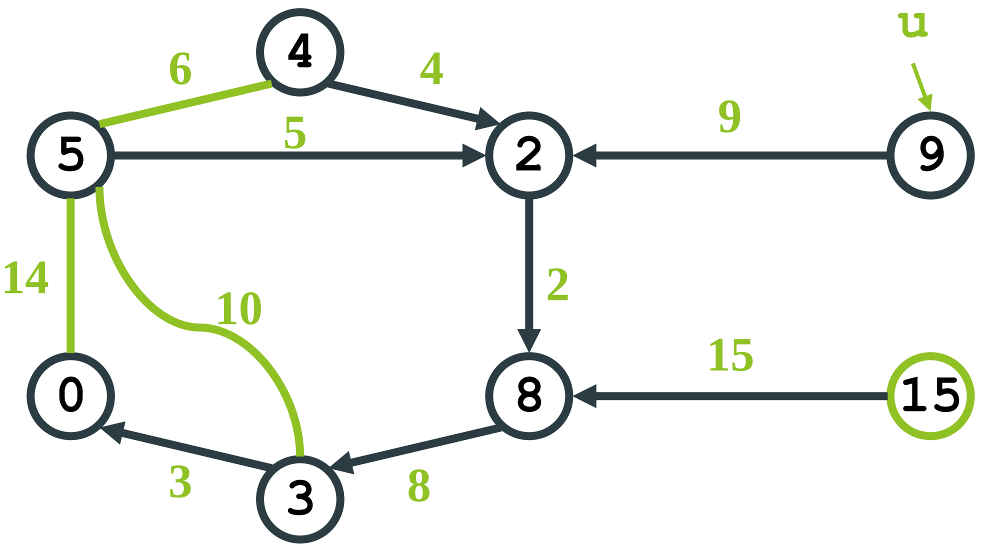
Prim’s Algorithm

```
MST-Prim(G, w, r)
    Q = G.V;
    for each u ∈ Q
        u.key = ∞;
    r.key = 0;
    r.p = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj(u)
            if (v ∈ Q and w(u,v) < v.key)
                v.p = u;
                v.key = w(u,v);
```



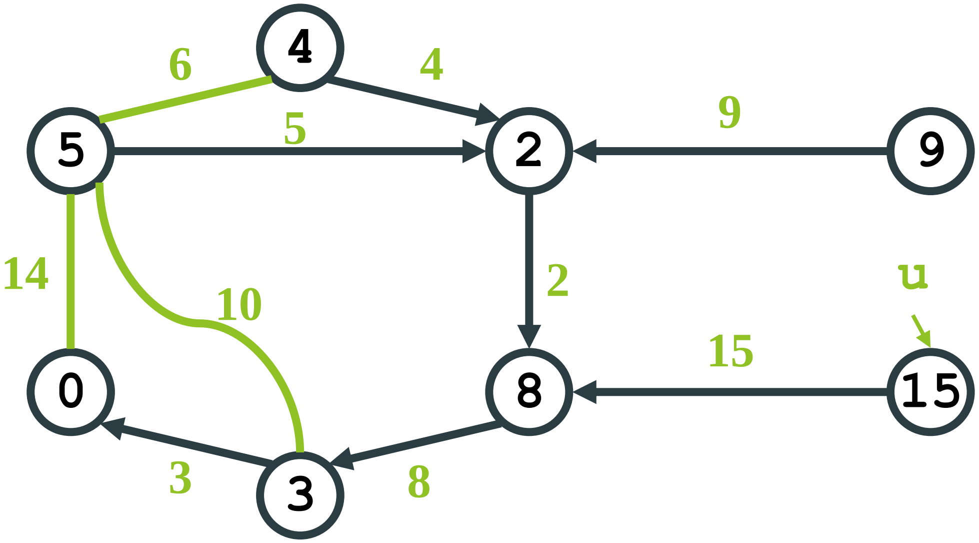
Prim’s Algorithm

```
MST-Prim(G, w, r)
    Q = G.V;
    for each u ∈ Q
        u.key = ∞;
    r.key = 0;
    r.p = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj(u)
            if (v ∈ Q and w(u,v) < v.key)
                v.p = u;
                v.key = w(u,v);
```

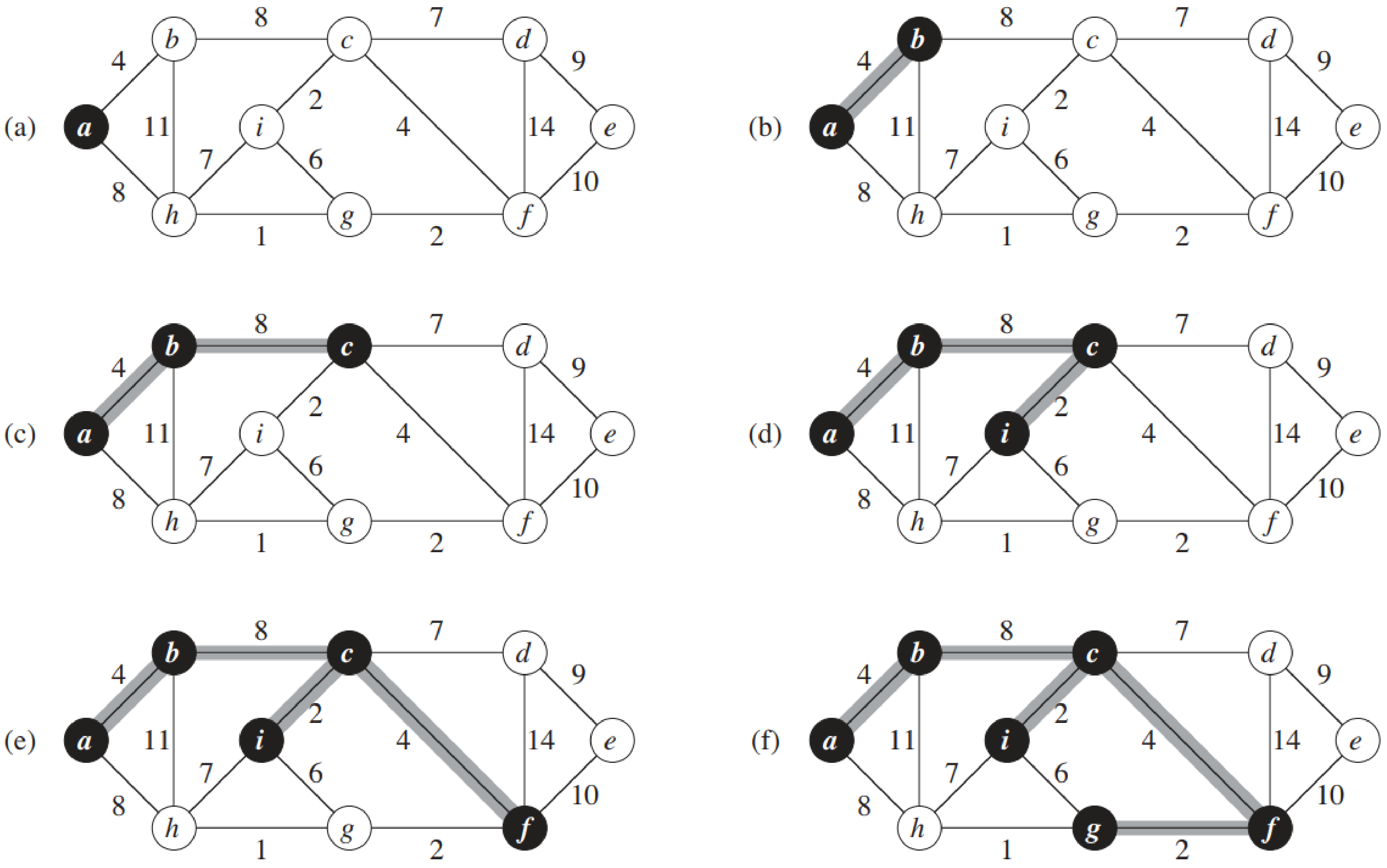


Prim’s Algorithm

```
MST-Prim(G, w, r)
    Q = G.V;
    for each u ∈ Q
        u.key = ∞;
    r.key = 0;
    r.p = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj(u)
            if (v ∈ Q and w(u,v) < v.key)
                v.p = u;
                v.key = w(u,v);
```



Prim’s Algorithm



Prim's Algorithm

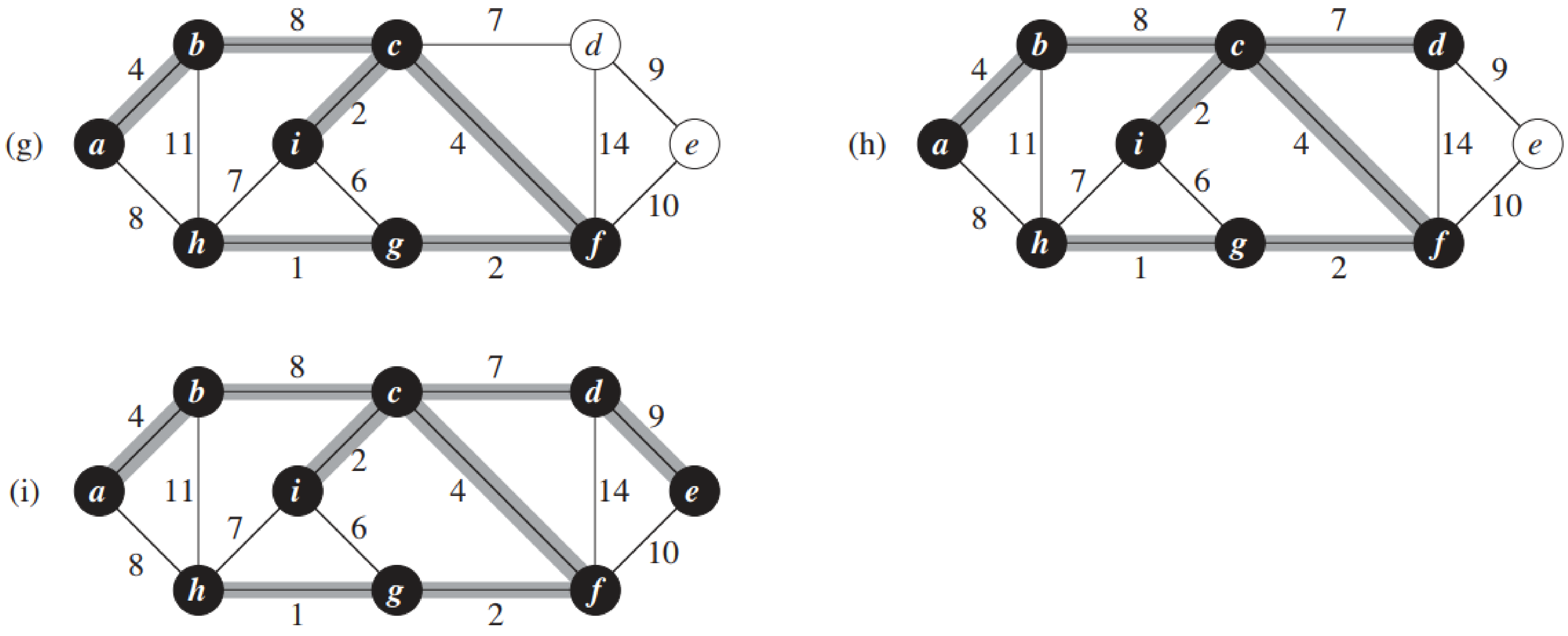


Figure 23.5 The execution of Prim's algorithm on the graph from Figure 23.1. The root vertex is a. Shaded edges are in the tree being grown, and black vertices are in the tree. At each step of the algorithm, the vertices in the tree determine a cut of the graph, and a light edge crossing the cut is added to the tree. In the second step, for example, the algorithm has a choice of adding either edge (b, c) or edge (a, h) to the tree since both are light edges crossing the cut.

Thanks to contributors

Mr. Pham Van Nguyen (2022)

Dr. Thien-Binh Dang (2017 – 2022)

Prof. Hyunseung Choo (2001 – 2022)