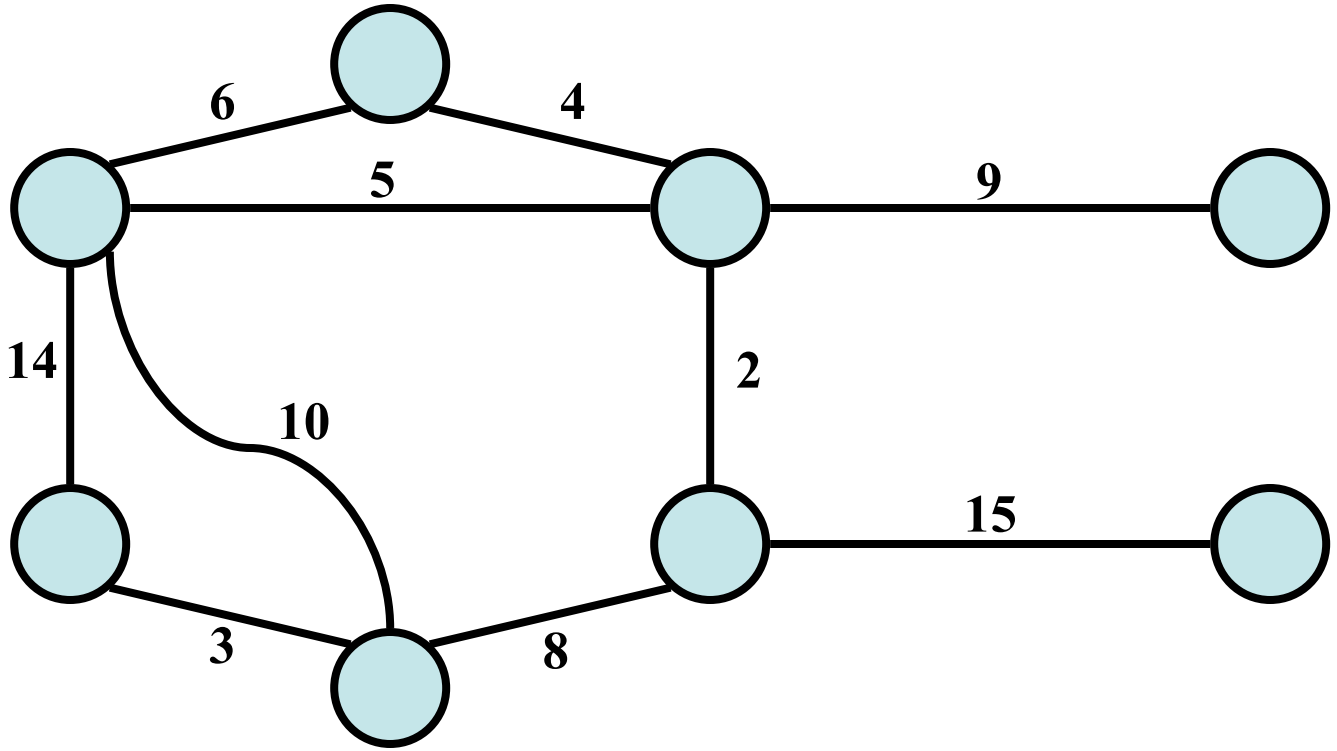
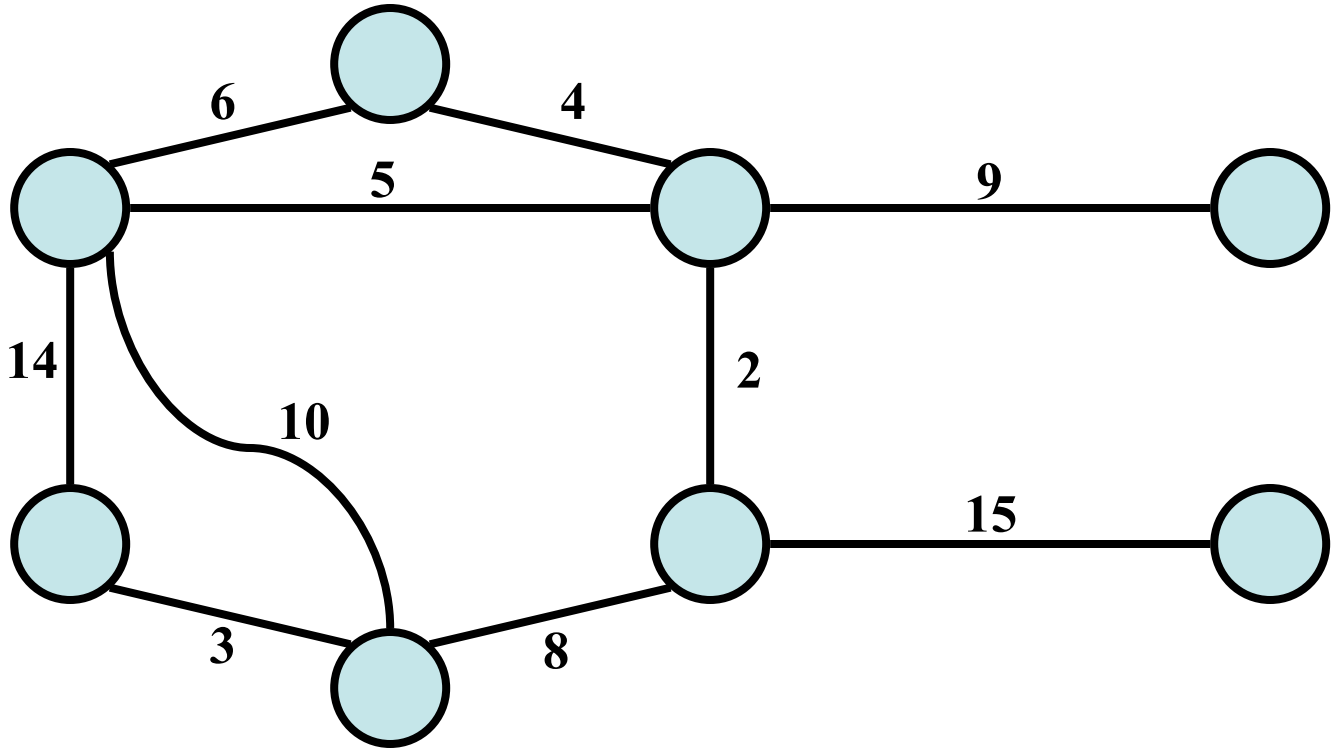
Minimum Spanning Tree

- Problem: given a connected, undirected, weighted graph:



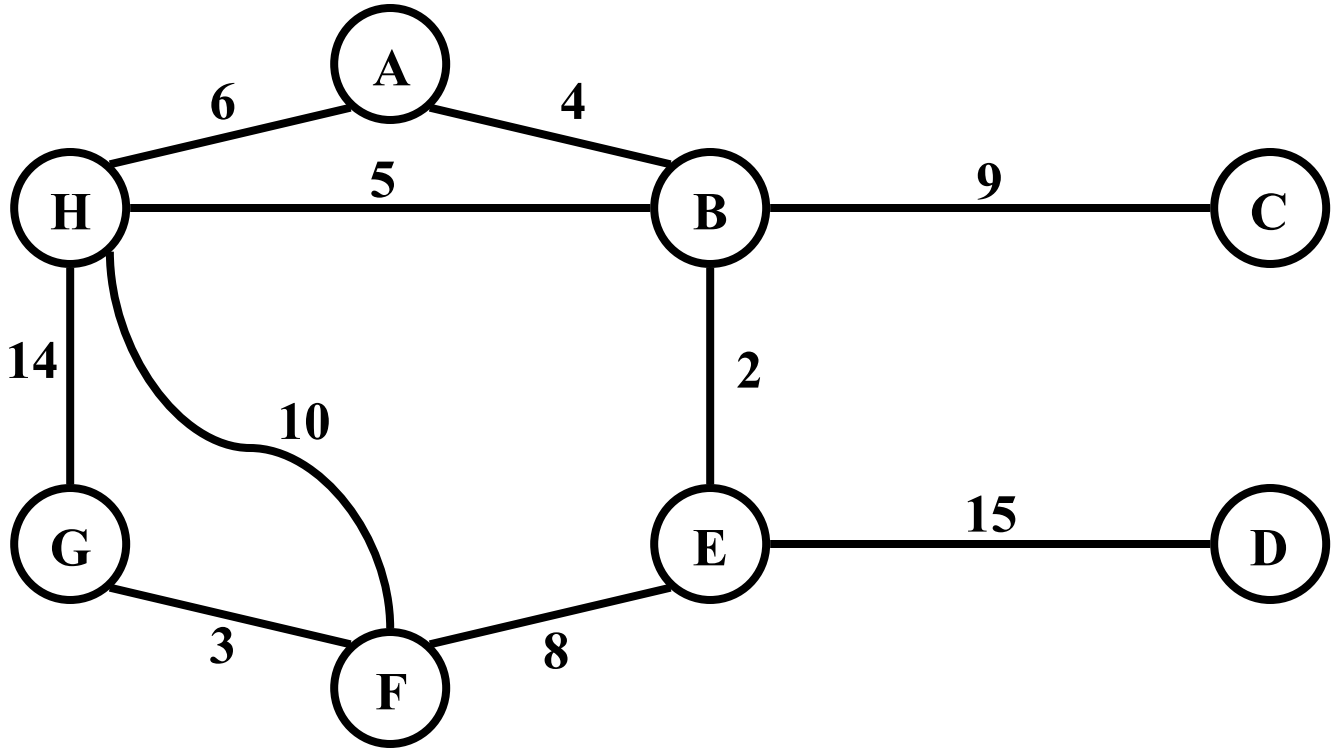
Minimum Spanning Tree

- Problem: given a connected, undirected, weighted graph, find a *spanning tree* using edges that minimize the total weight



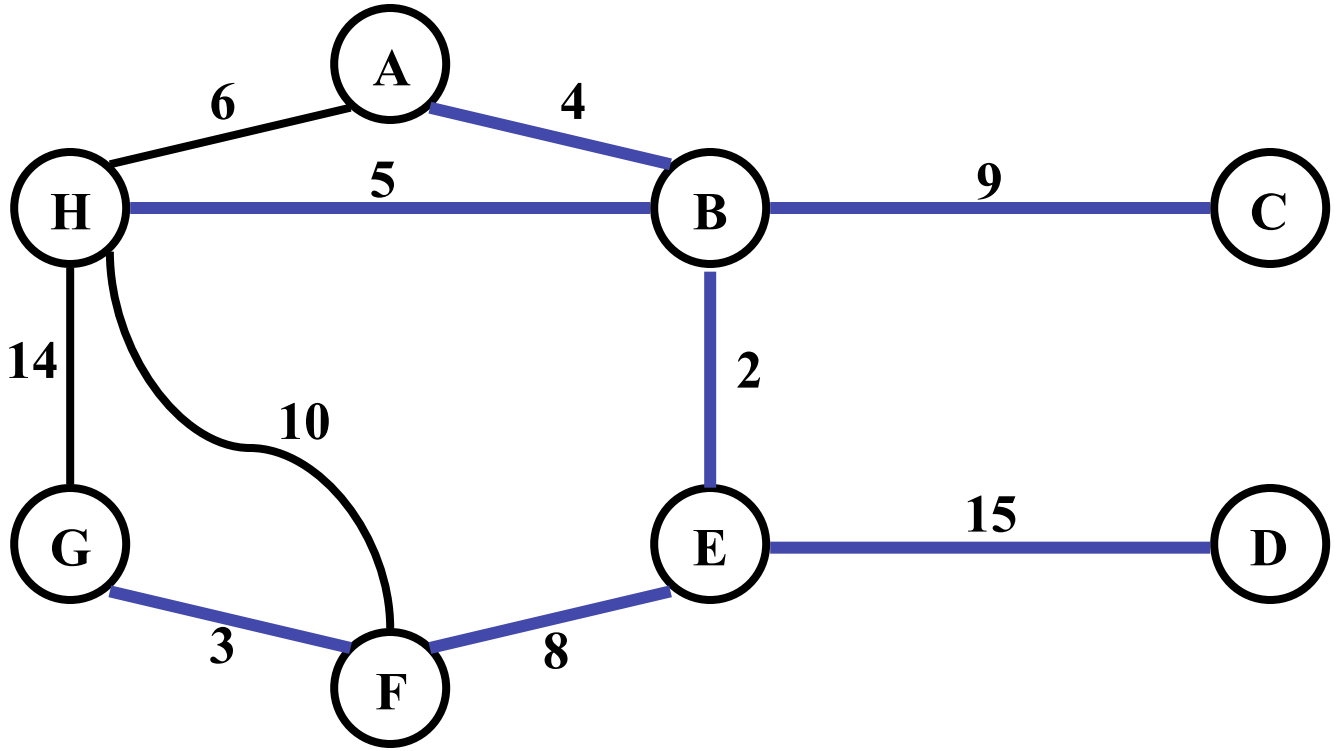
Minimum Spanning Tree

- *Which edges form the minimum spanning tree (MST) of the below graph?*



Minimum Spanning Tree

- Answer:



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)

Minimum Spanning Tree

- Thm:
  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

Minimum Spanning Tree

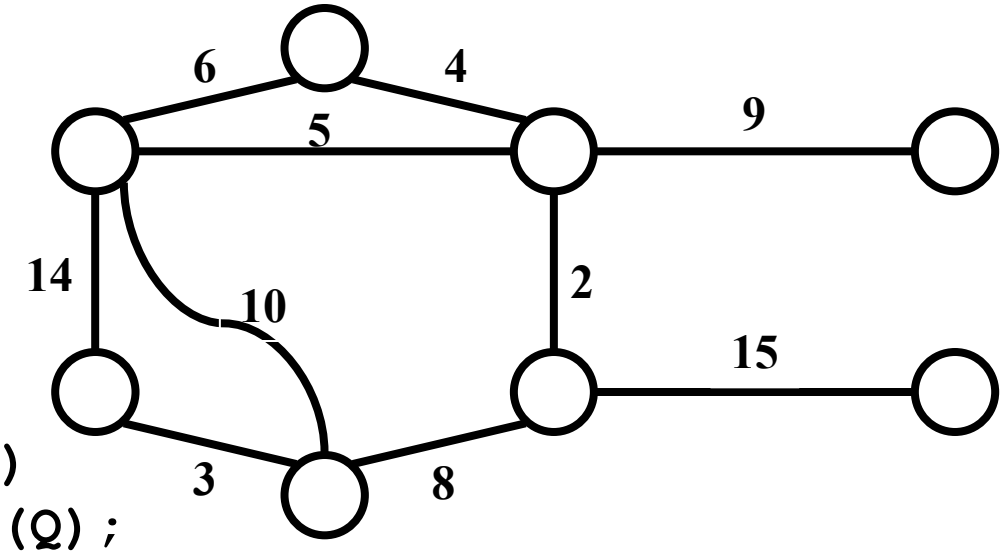
- Thm:
  Let T be MST of G, and let A $\subseteq$ T be subtree of T
  Let (u,v) be min-weight edge connecting A to V-A
  Then $(u,v) \in$ T
- Proof: in book (see Thm 23.1)

Prim's Algorithm

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

Prim's Algorithm

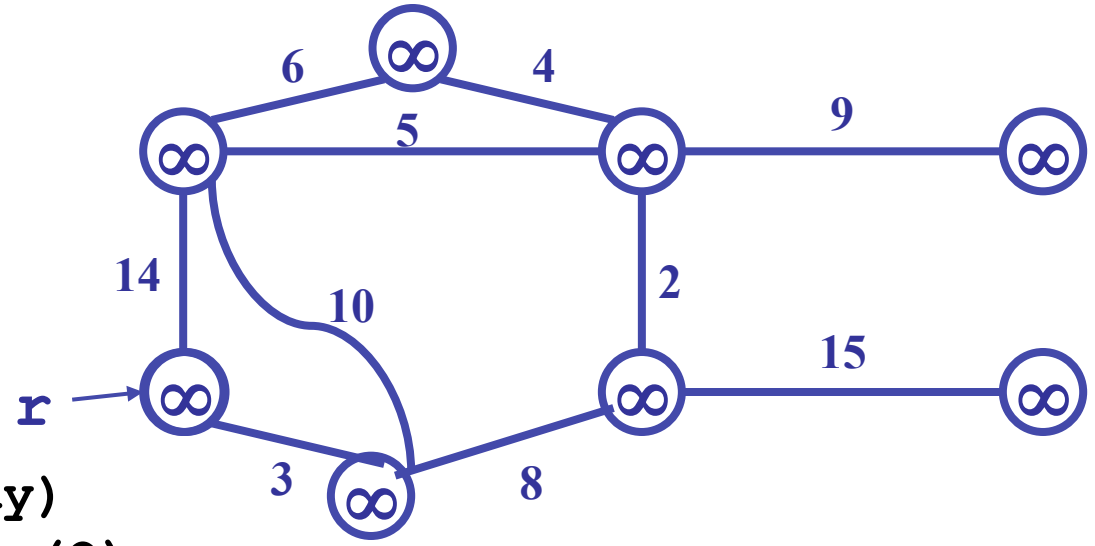
```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



Run on example graph

Prim's Algorithm

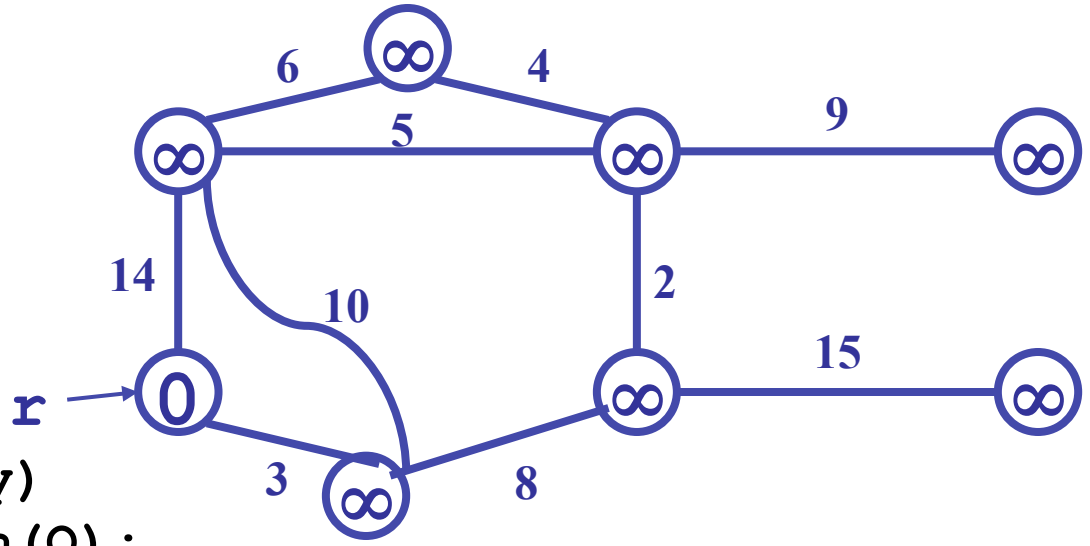
```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



Pick a start vertex r

Prim's Algorithm

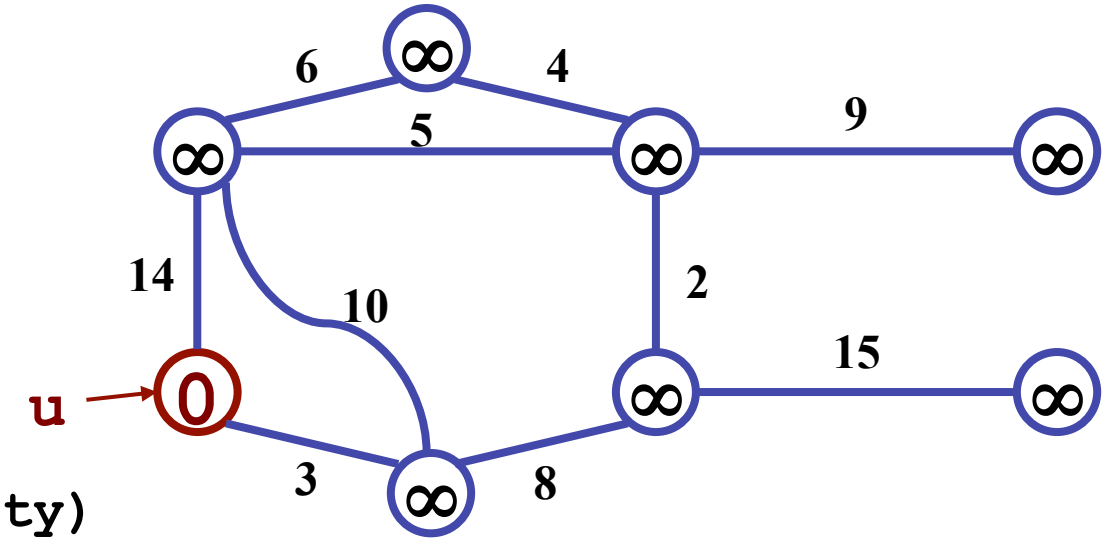
```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



Pick a start vertex r

Prim's Algorithm

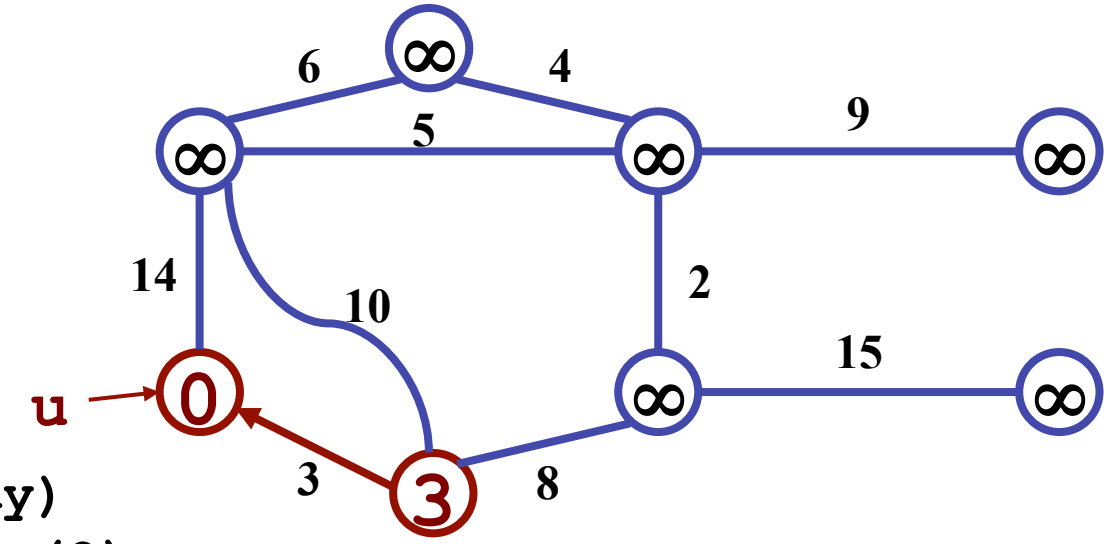
```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



Red vertices have been removed from Q

Prim's Algorithm

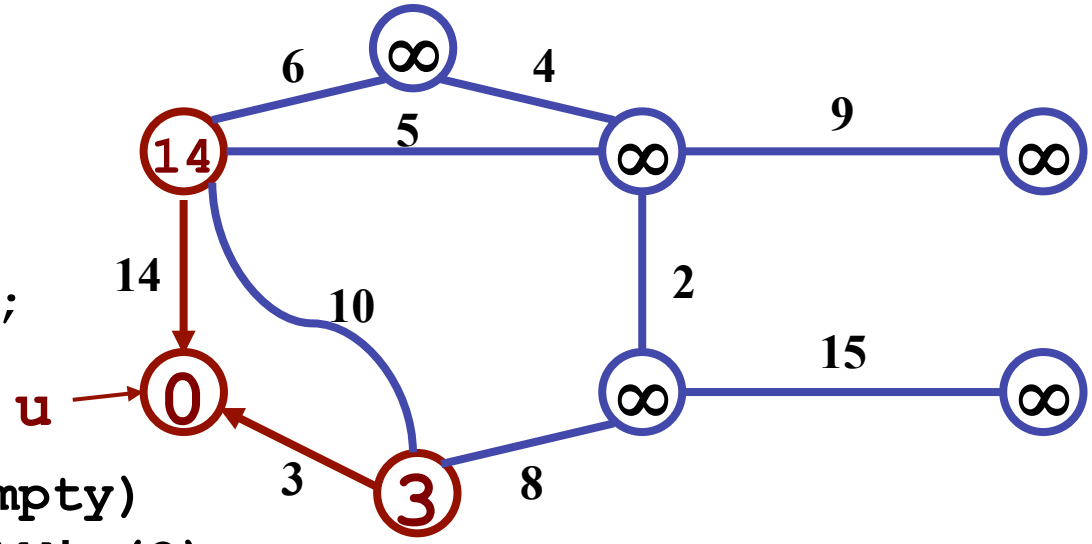
```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



Red arrows indicate parent pointers

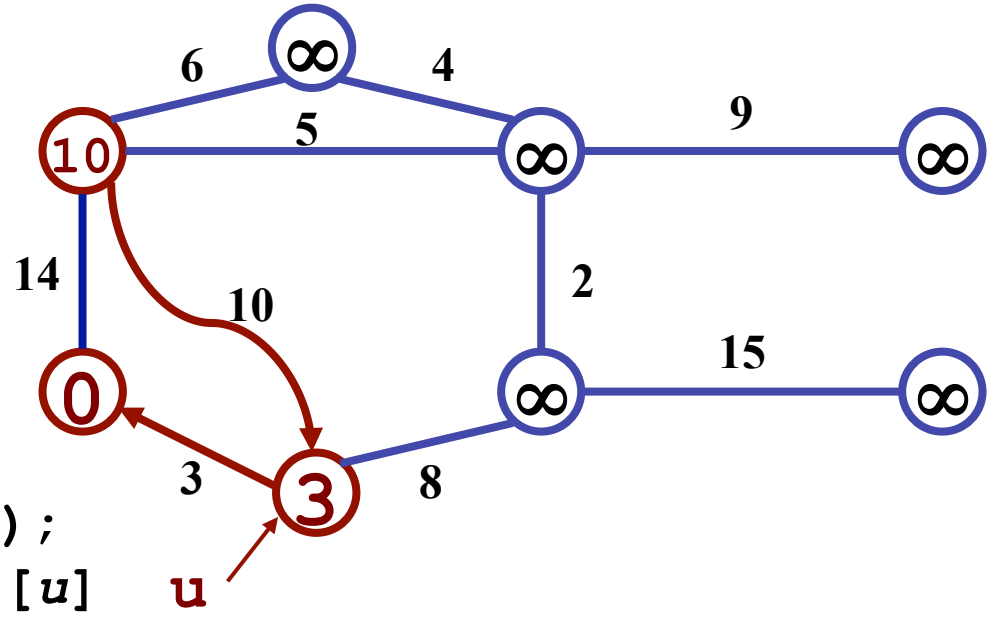
Prim's Algorithm

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



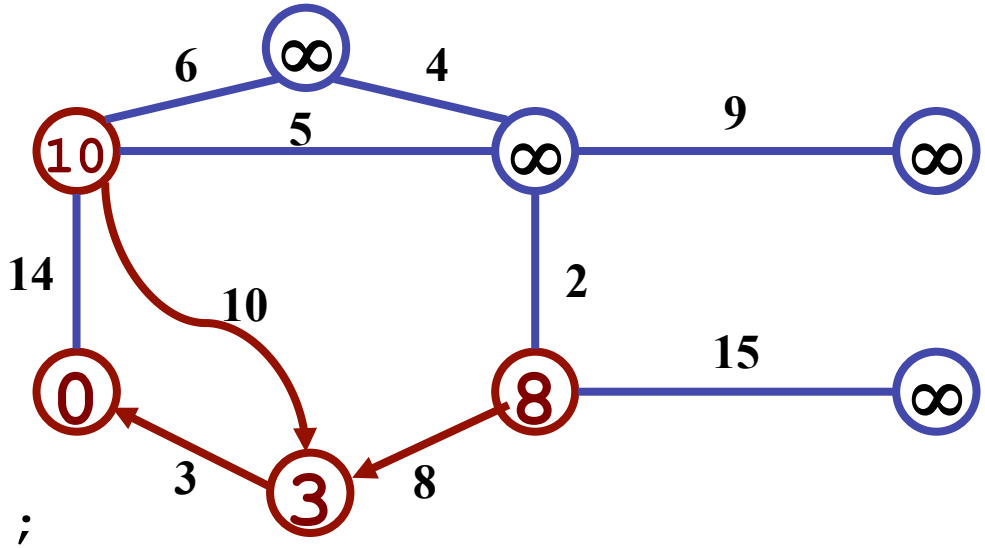
Prim's Algorithm

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



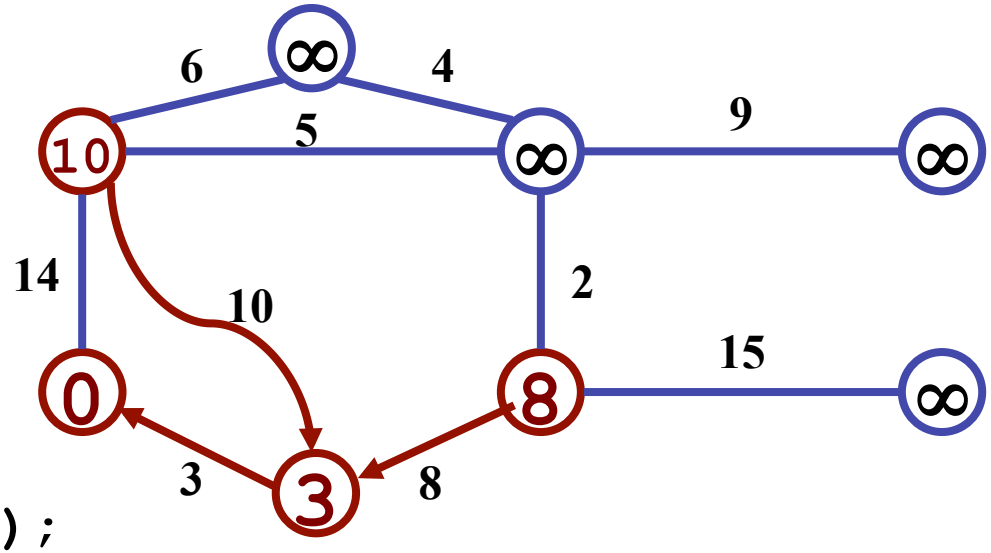
Prim's Algorithm

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



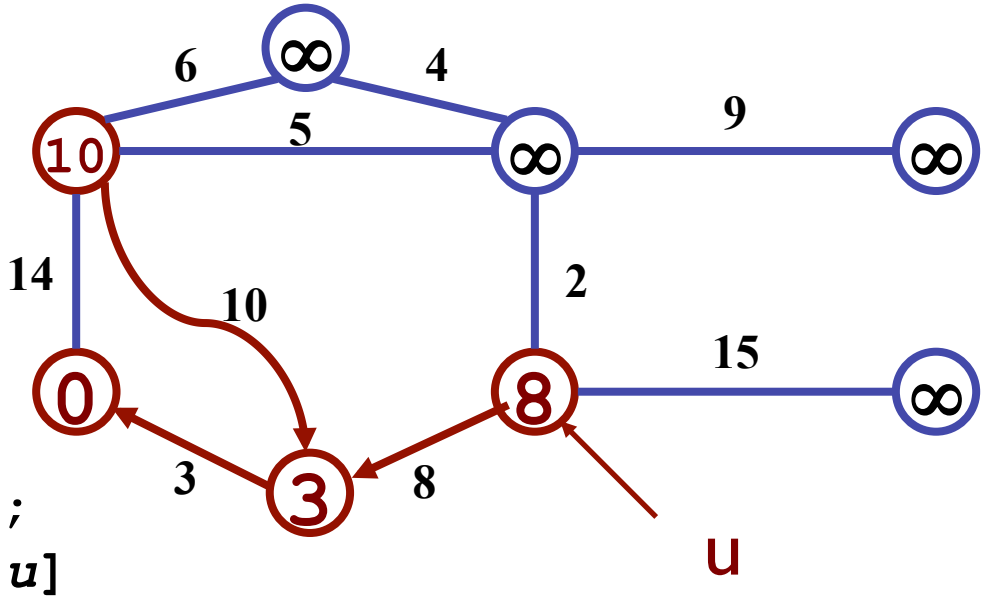
Prim's Algorithm

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



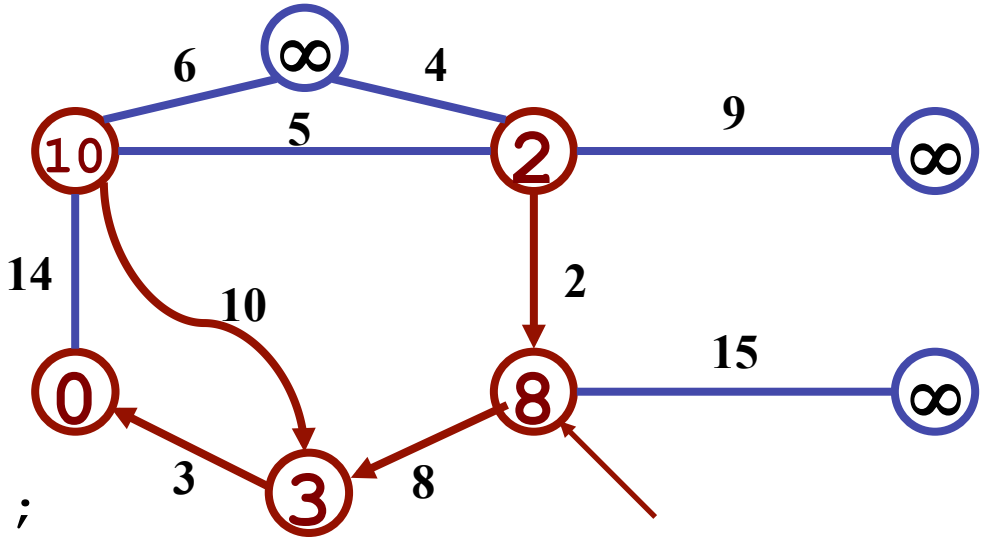
Prim's Algorithm

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



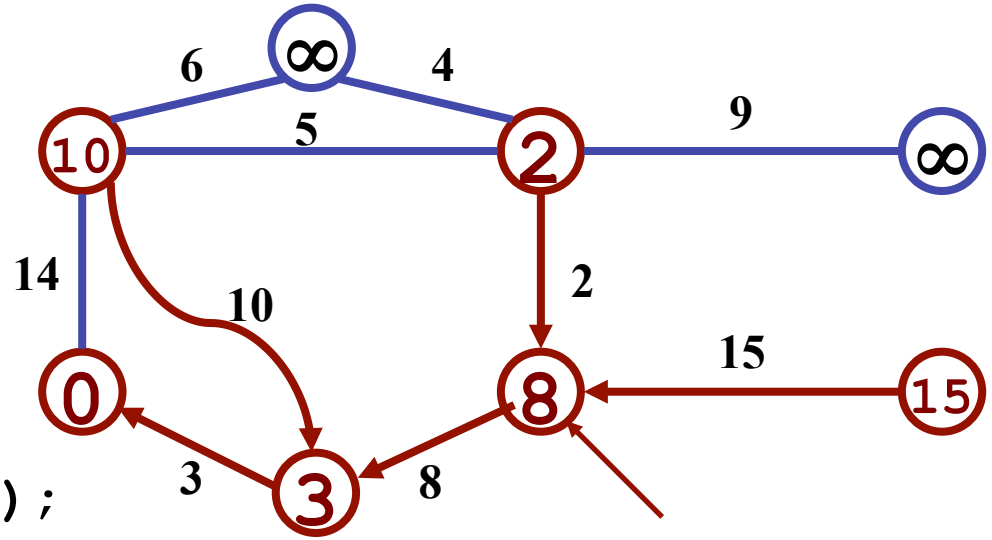
Prim's Algorithm

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



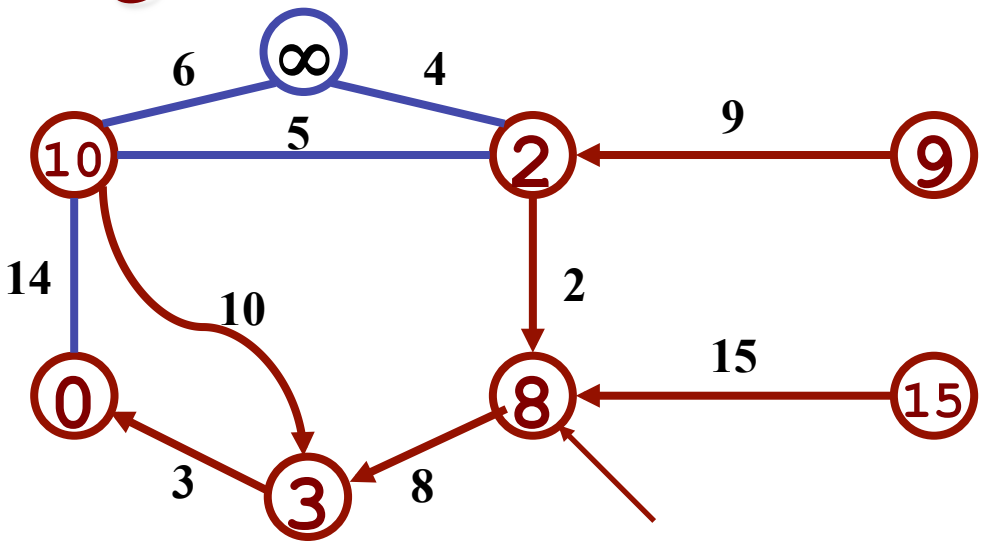
Prim's Algorithm

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



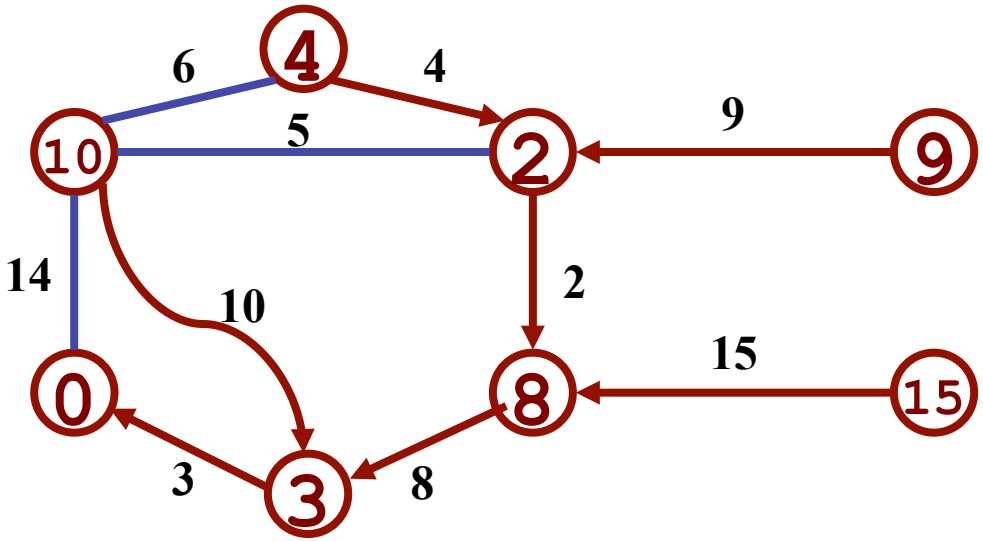
Prim's Algorithm

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



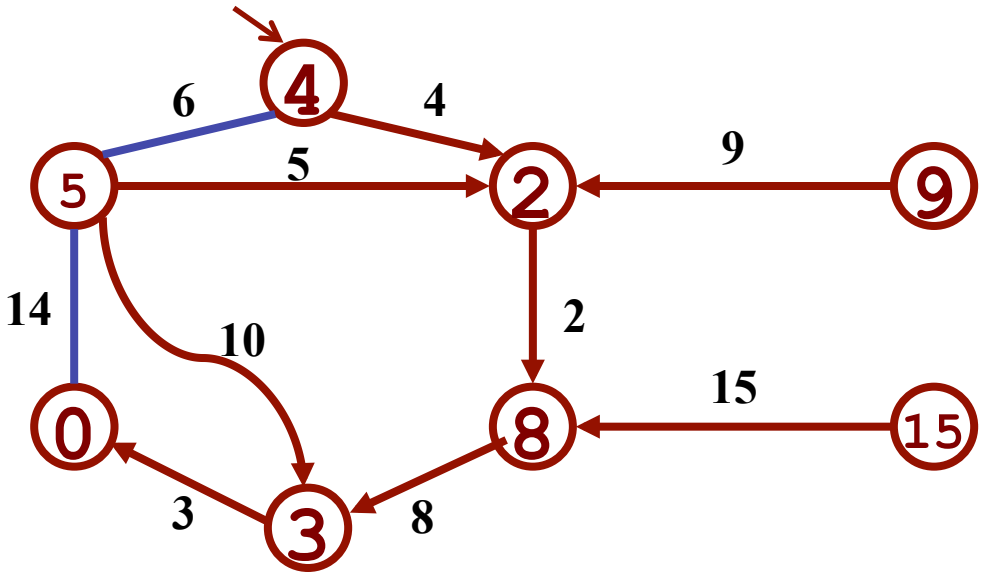
Prim's Algorithm

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



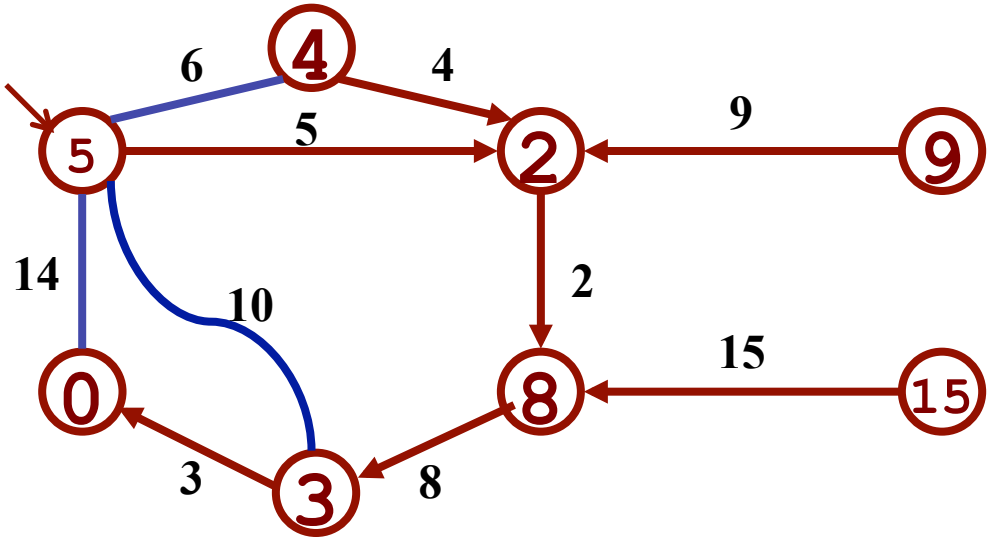
Prim's Algorithm

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



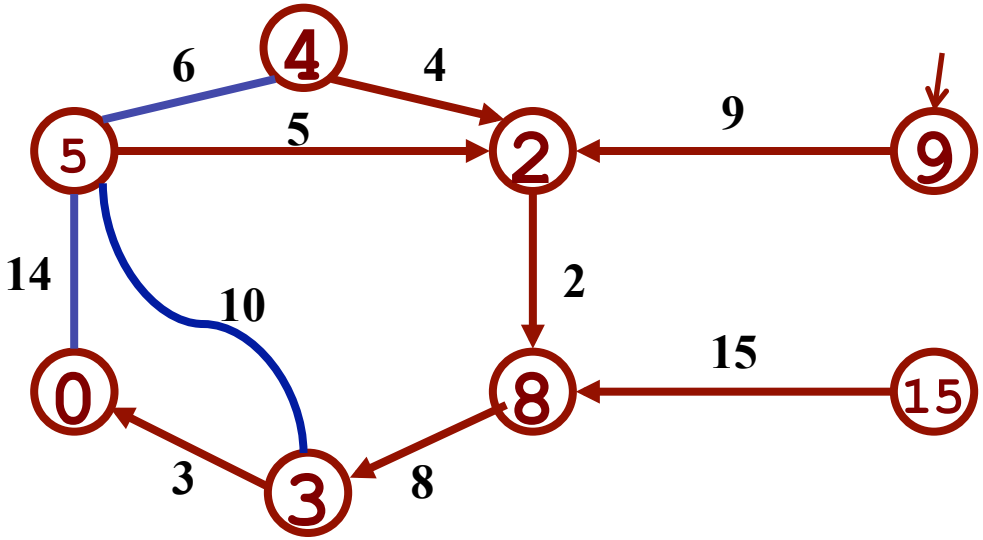
Prim's Algorithm

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



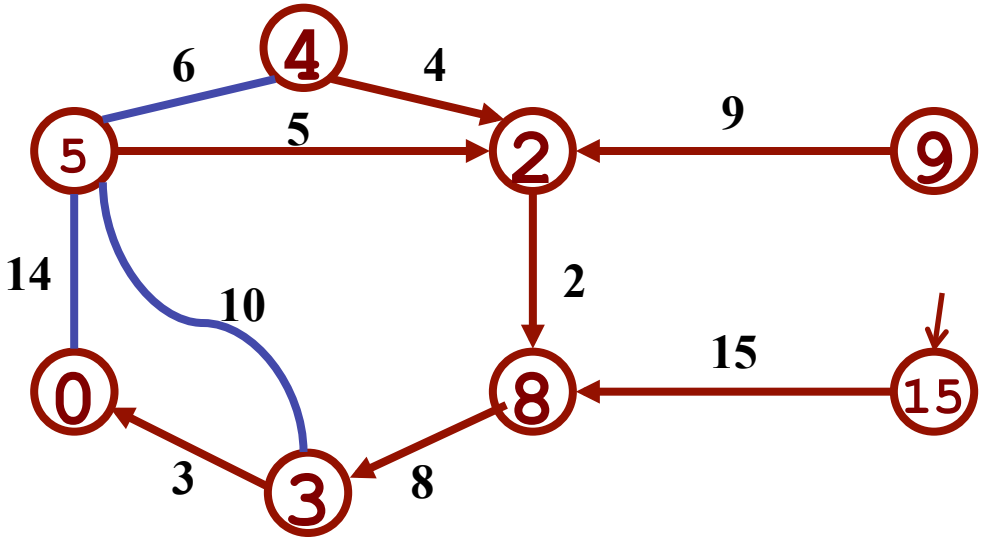
Prim's Algorithm

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



Prim's Algorithm

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



Review: Prim's Algorithm

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

Review: Prim's Algorithm

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

How often is ExtractMin() called?

How often is DecreaseKey() called?

ExtractMin total number of calls O(V log V)

DecreaseKey total number of calls O(E log V)

Review: Prim's Algorithm

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

What will be the running time?
A: Depends on queue
binary heap: O(E lg V)
Fibonacci heap: O(V lg V + E)

ExtractMin total number of calls O(V log V)

DecreaseKey total number of calls O(E log V)

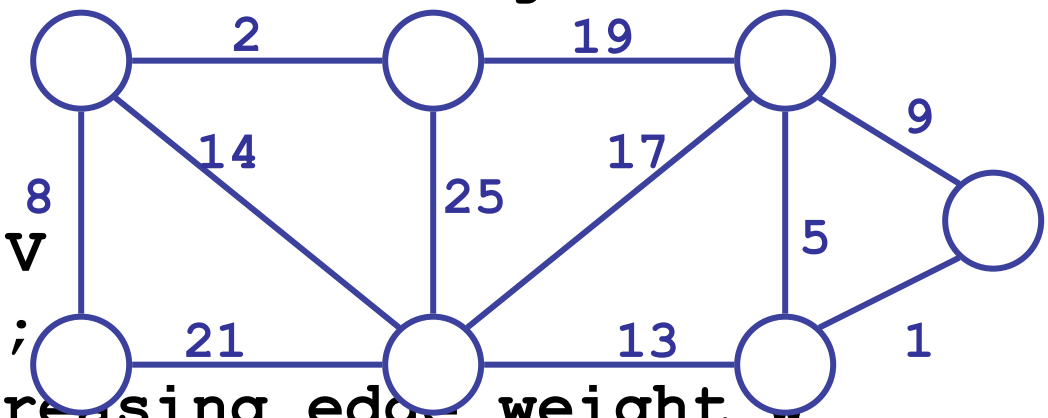
Total number of calls O(V logV +E logV) = O(E log V)

Think why we can combine things in the expression above

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));
}
```

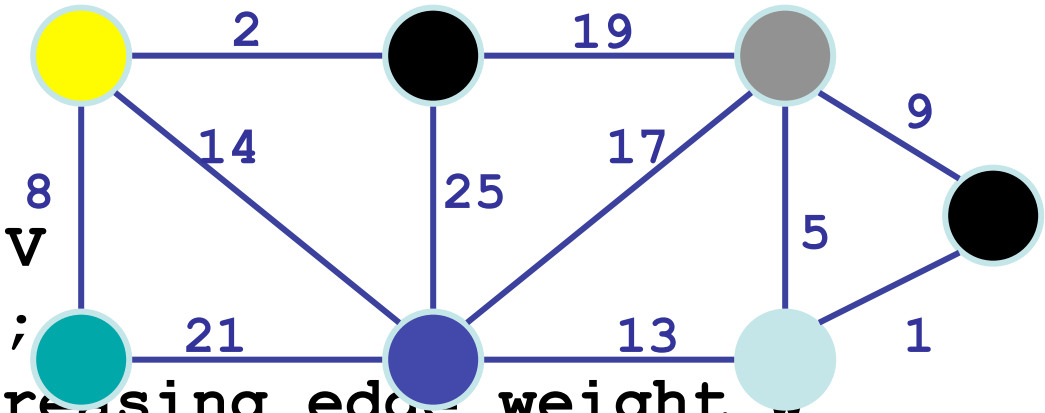
Run the algorithm:



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));
}
```

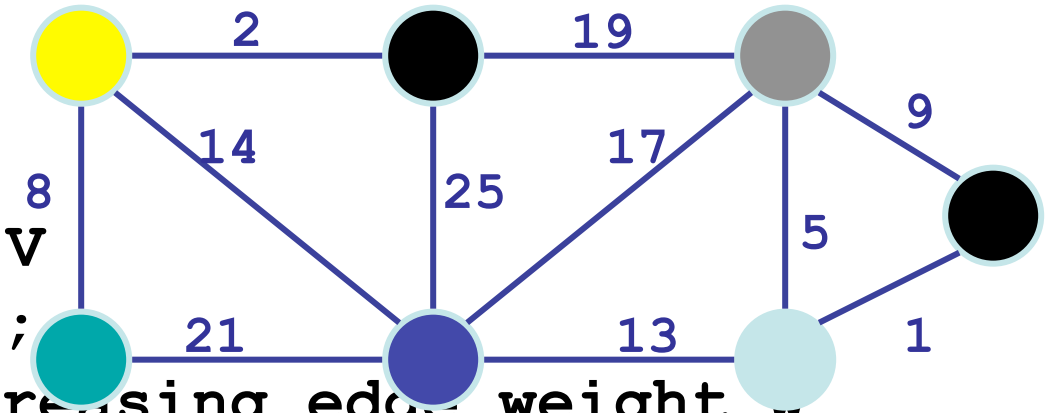
Run the algorithm:



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));
}
```

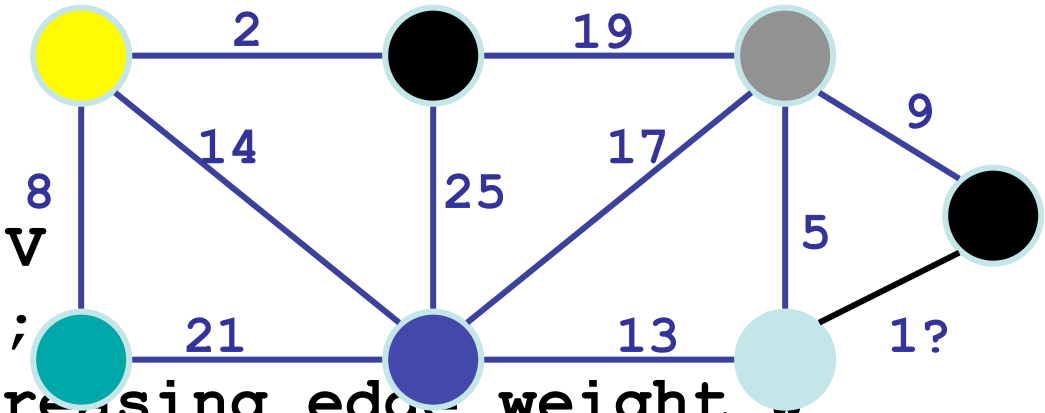
Run the algorithm:



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));
}
```

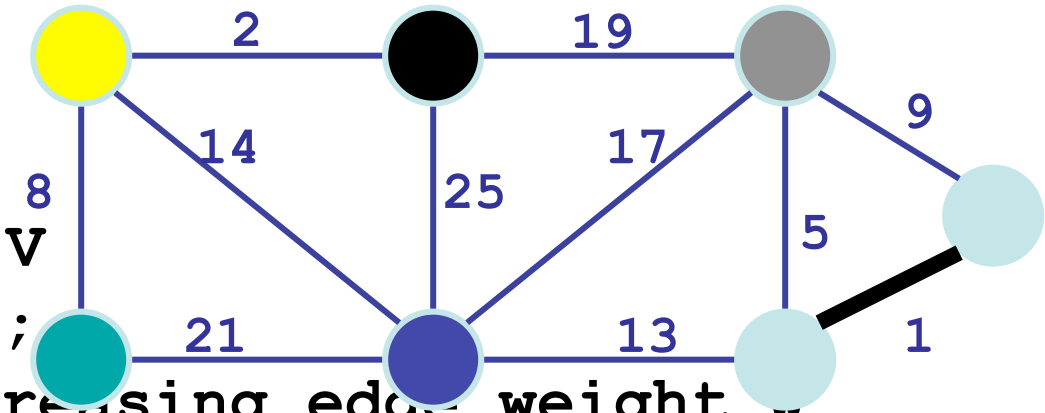
Run the algorithm:



Kruskal's Algorithm

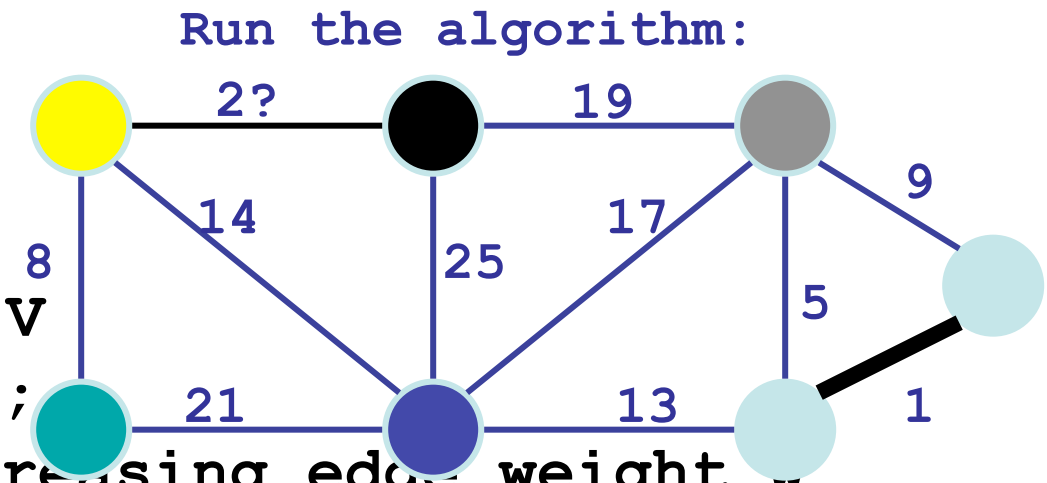
```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

Run the algorithm:



Kruskal's Algorithm

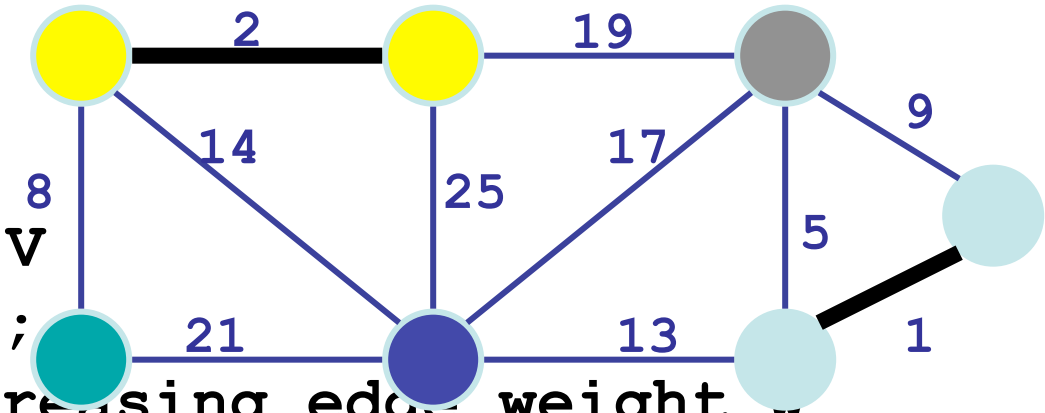
```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```



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));
}
```

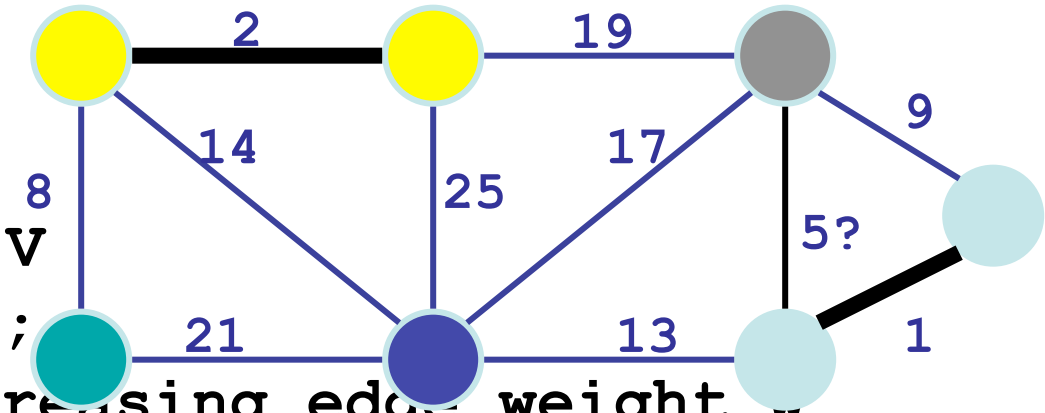
Run the algorithm:



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));
}
```

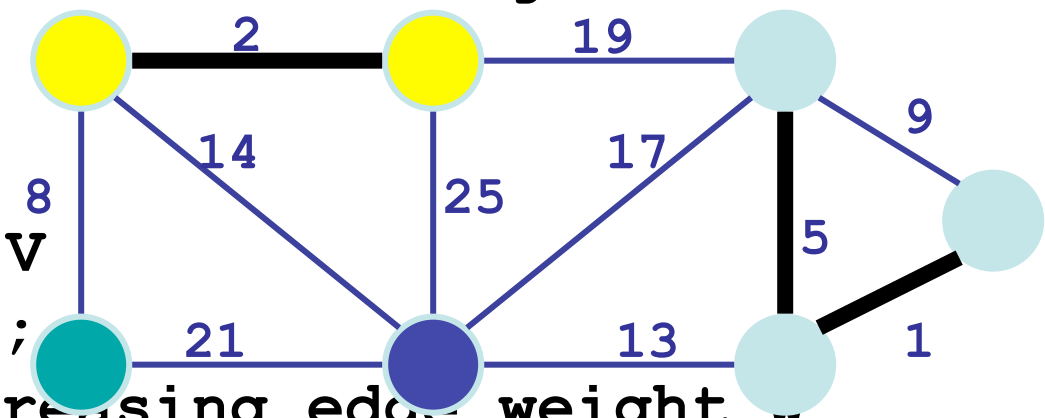
Run the algorithm:



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));
}
```

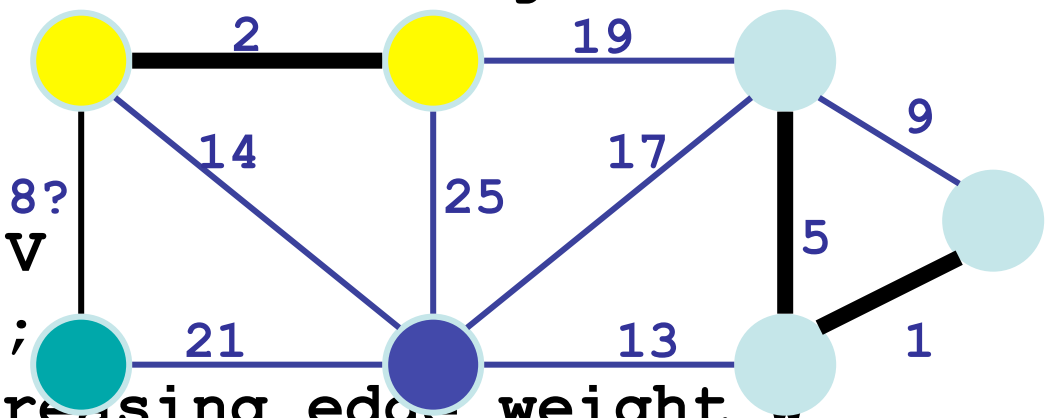
Run the algorithm:



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));
}
```

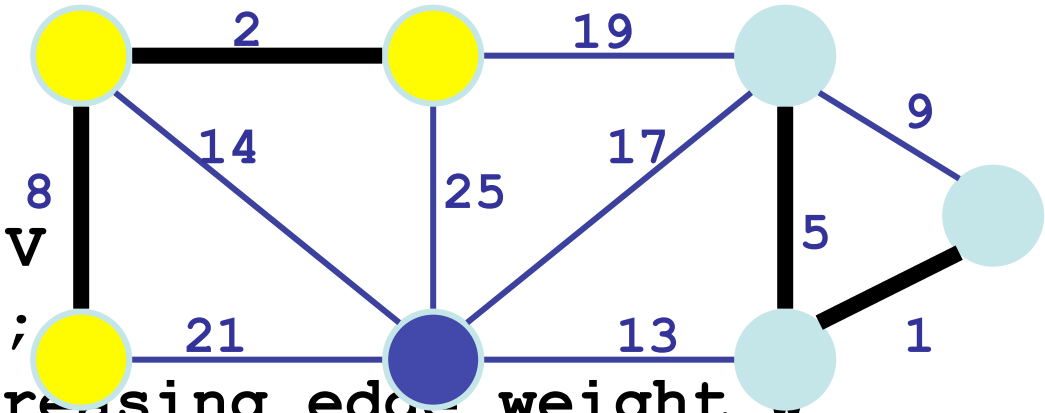
Run the algorithm:



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));
}
```

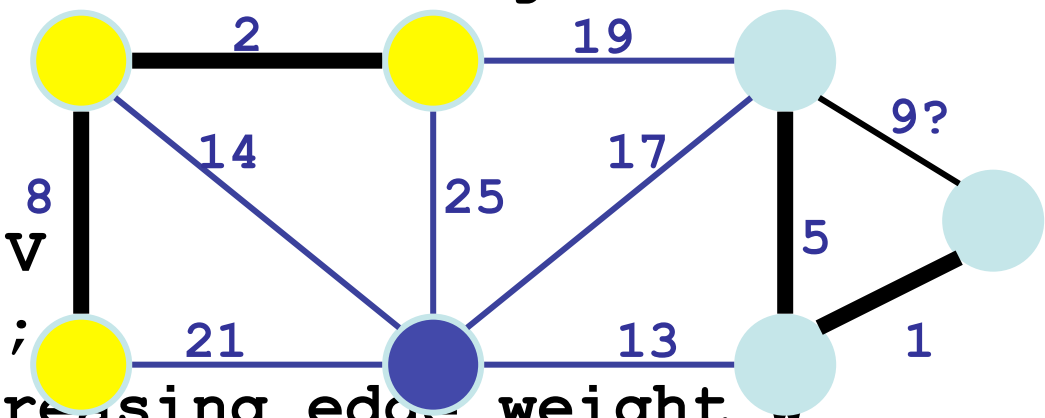
Run the algorithm:



Kruskal's Algorithm

```
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));
}
```

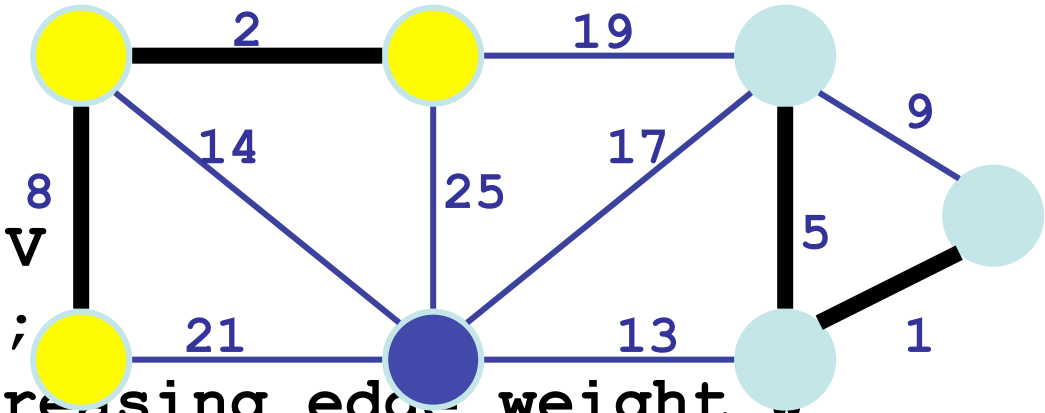
Run the algorithm:



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));
}
```

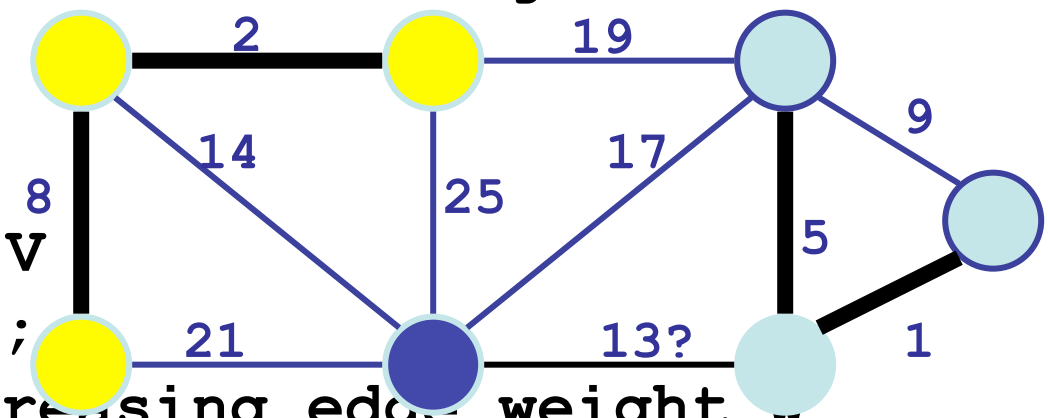
Run the algorithm:



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));
}
```

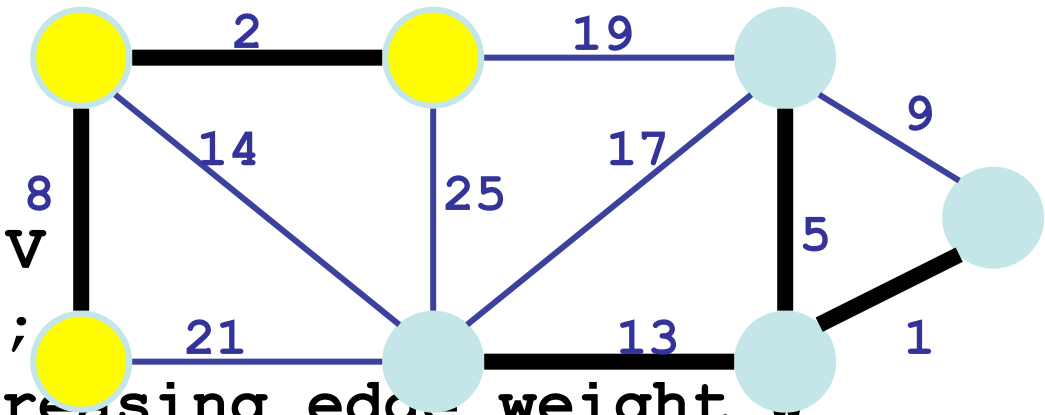
Run the algorithm:



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));
}
```

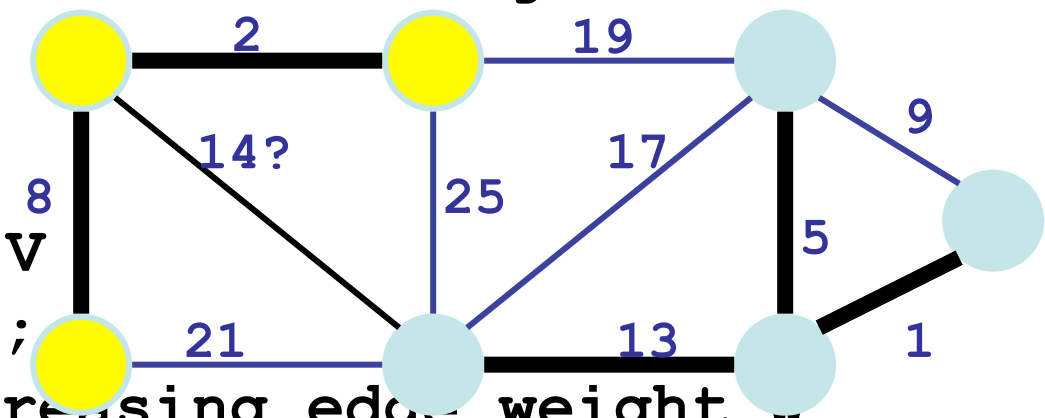
Run the algorithm:



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));
}
```

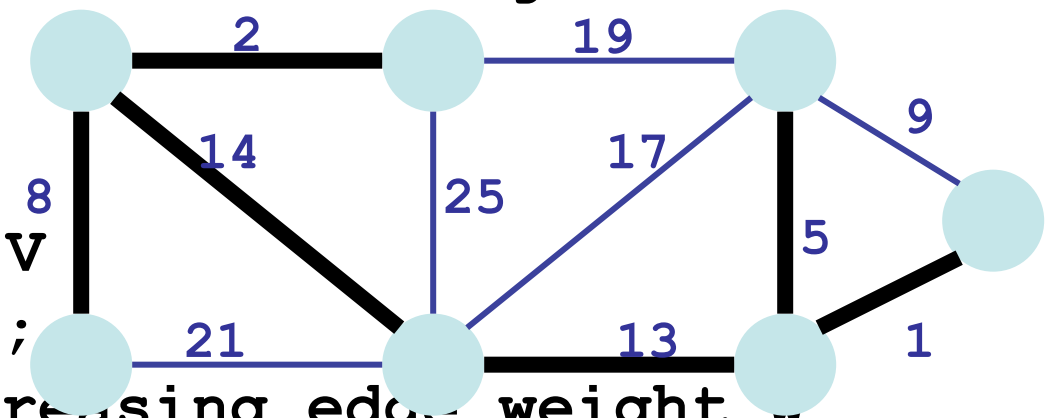
Run the algorithm:



Kruskal's Algorithm

```
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));
}
```

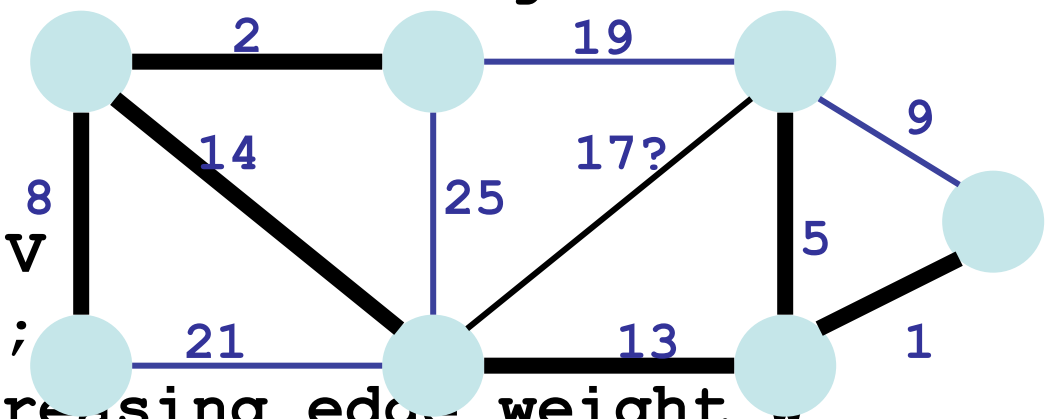
Run the algorithm:



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));
}
```

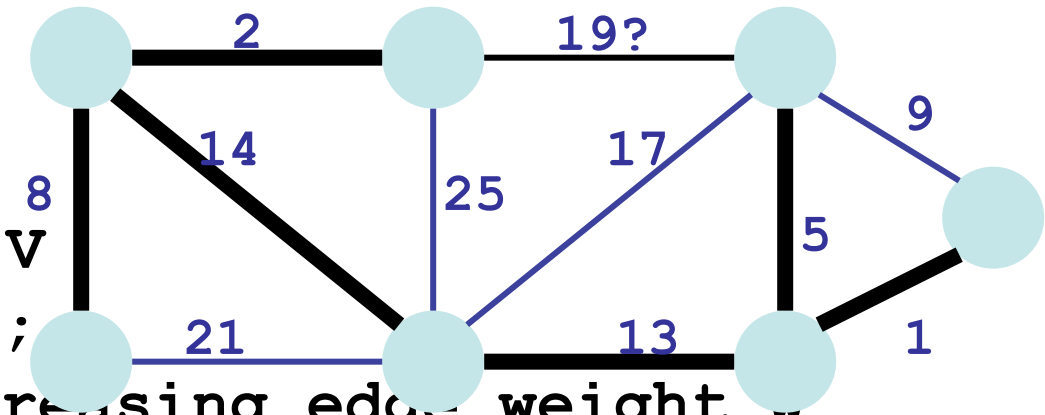
Run the algorithm:



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));
}
```

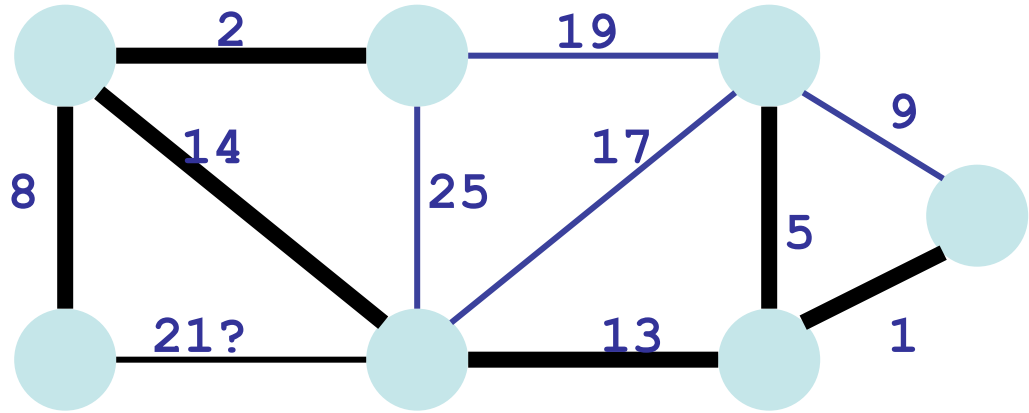
Run the algorithm:



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,v}};
            Union(FindSet(u), FindSet(v));
}
```

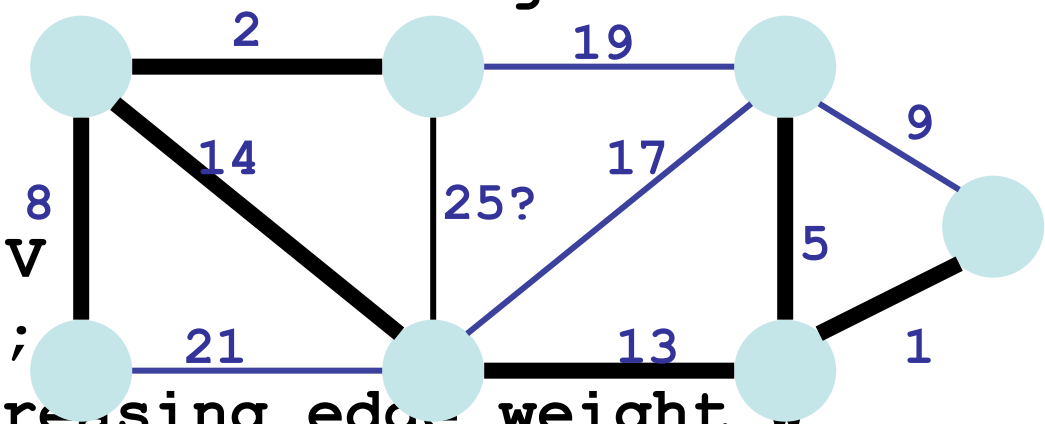
Run the algorithm:



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));
}
```

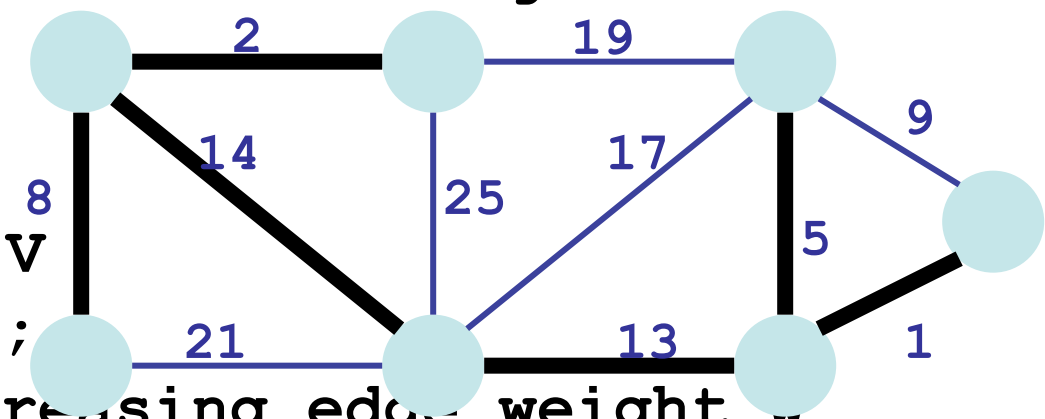
Run the algorithm:



Kruskal's Algorithm

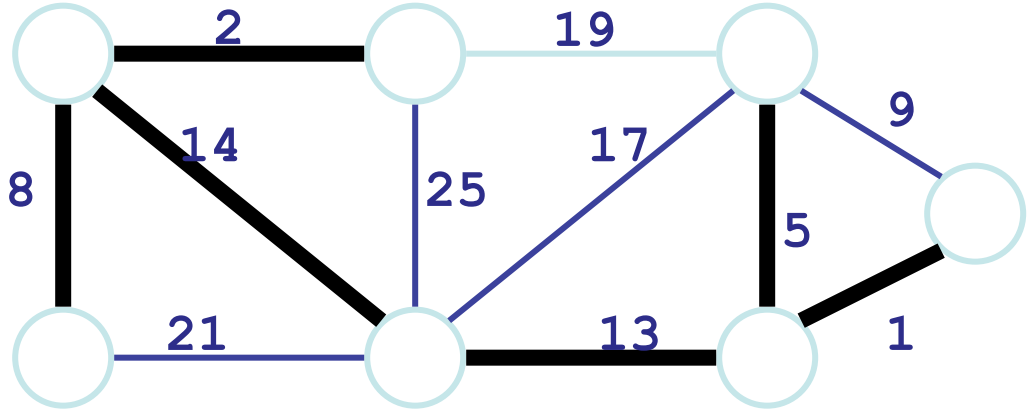
```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

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,v}};
            Union(FindSet(u), FindSet(v));
}
```