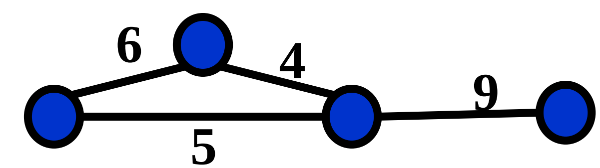
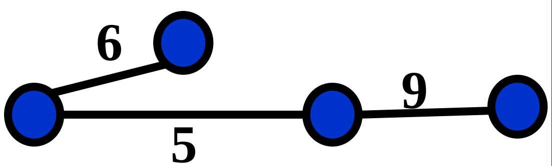
Lecture 10:
Minimum Spanning Tree (MST)

Spanning Tree and Minimum Spanning Tree

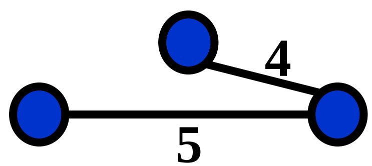
- **Spanning tree** of a connected graph: A tree that connects ***all*** nodes of the graph.
- **Minimum Spanning tree** of a connected weighted graph: A tree that connects all nodes of the graph with total edge weight lowest possible.
- **Example:** For the graph G,
  - **T1, T3 are spanning trees**
  - **T2 is not a spanning tree, because it does not connect all nodes of G**
  - **T1 is not minimum, because its total weight is 20**
  - **T2 is minimum with lowest possible total weight 19**



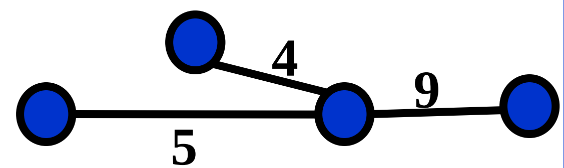
G



T1, weight: 6+5+9=20



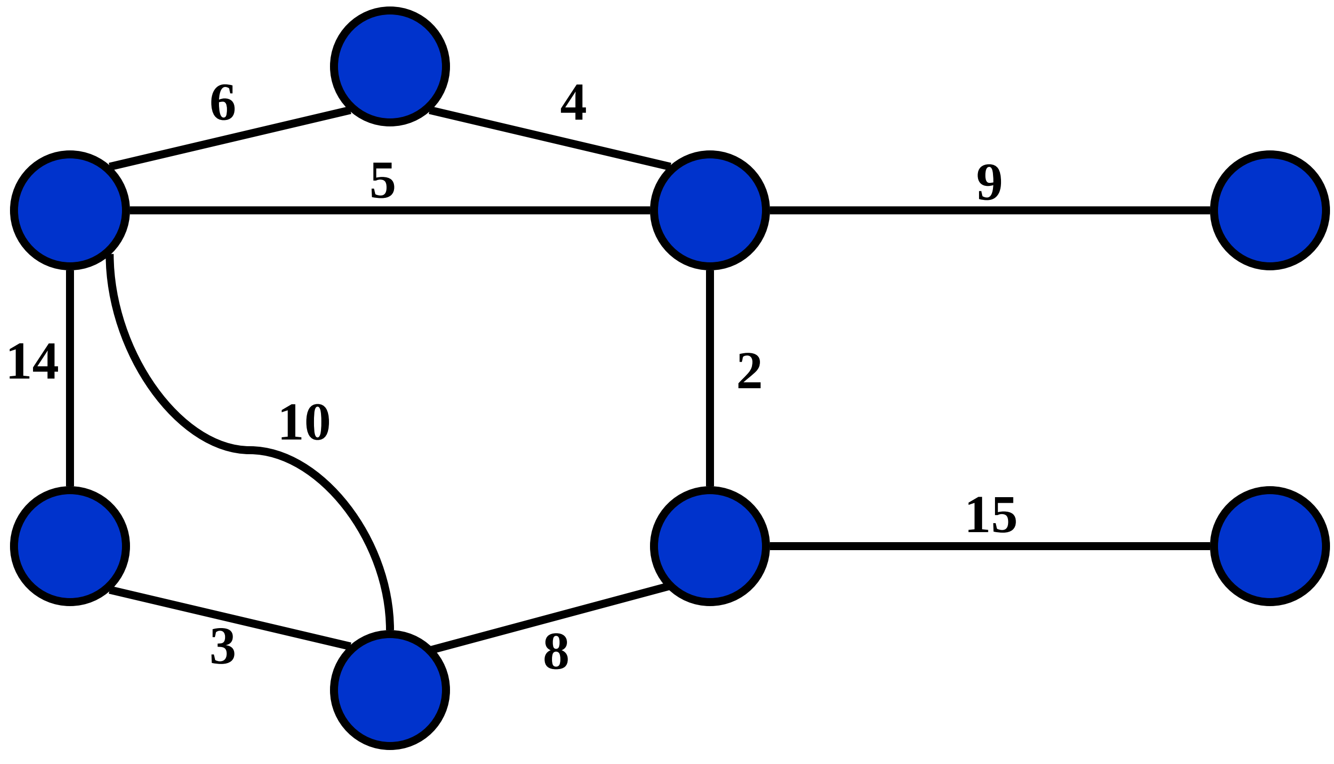
T2, not a spanning tree



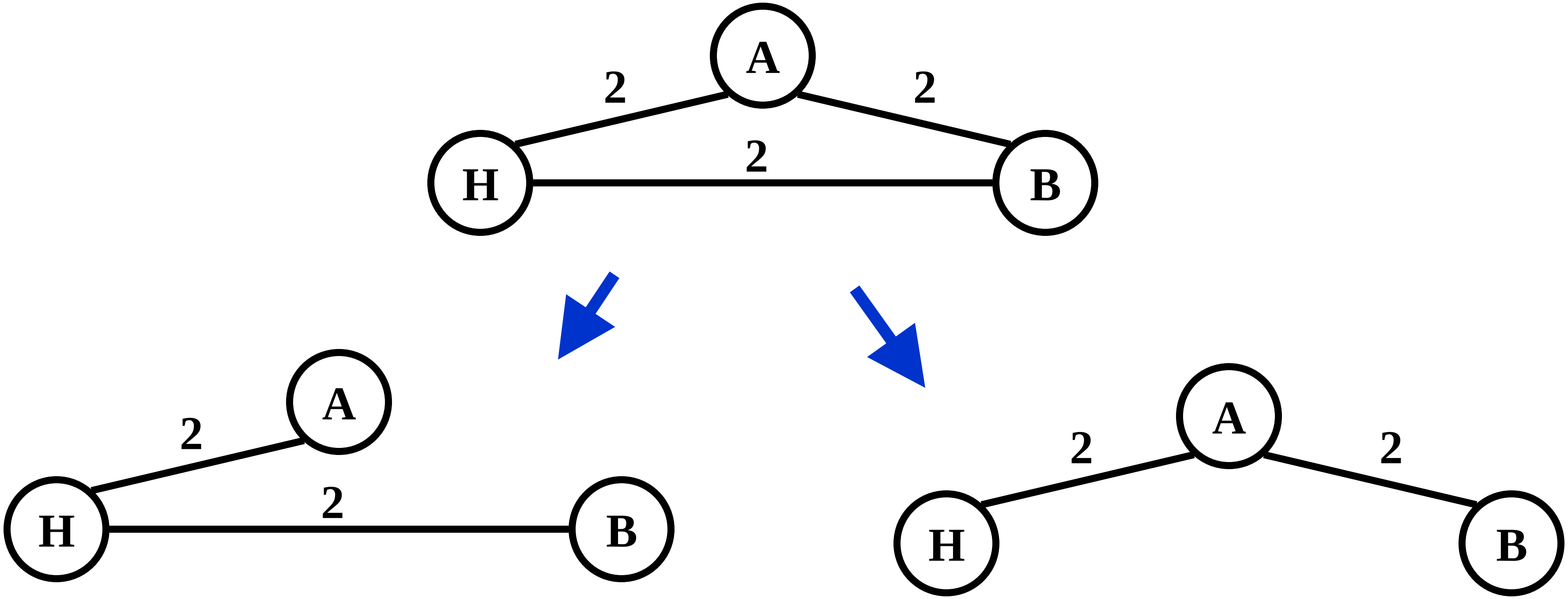
T3, weight: 5+4+9=18

Minimum Spanning Tree

- Problem: given a connected, undirected, weighted graph, find a *minimum spanning tree*

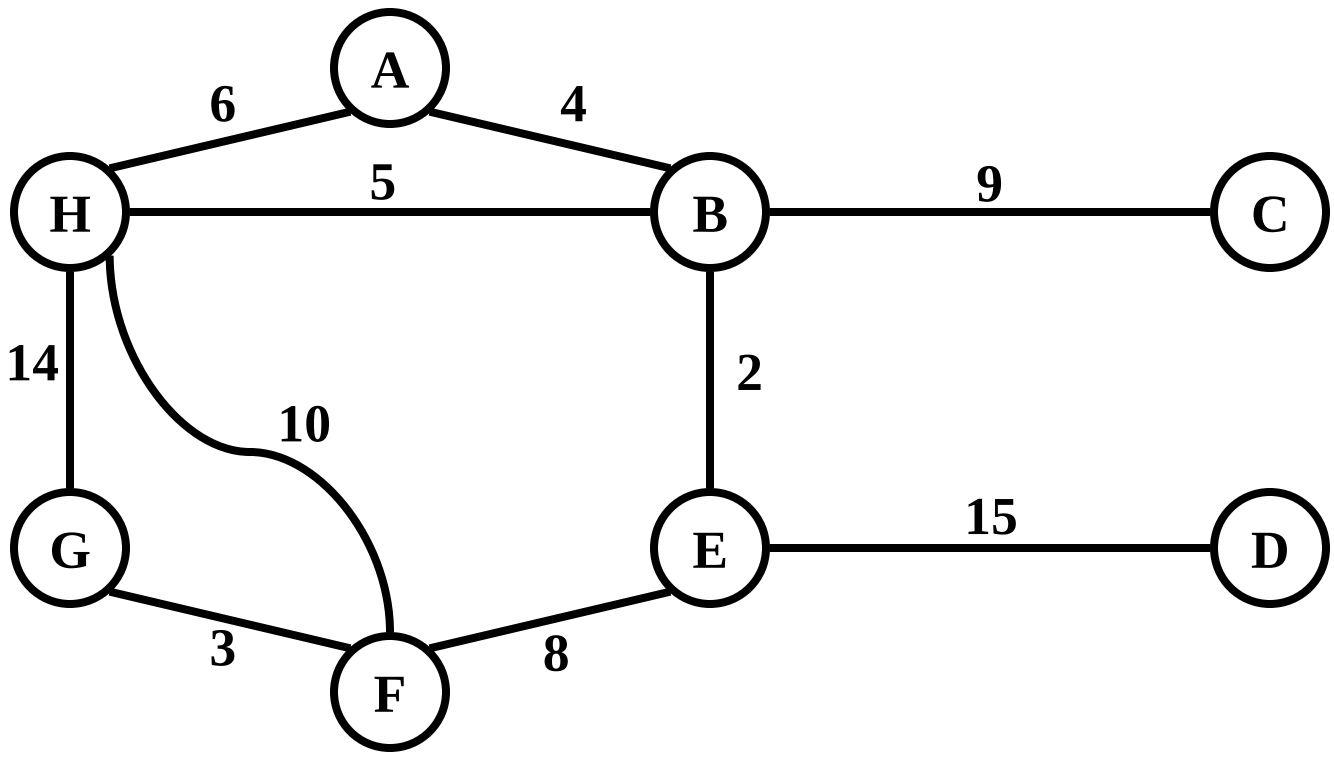


Mora Than One Minimum Spanning Tree Possible



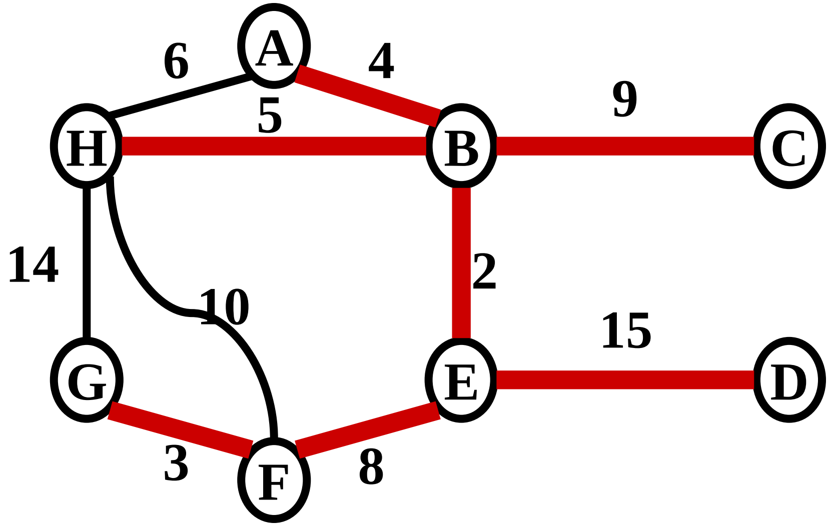
Minimum Spanning Tree

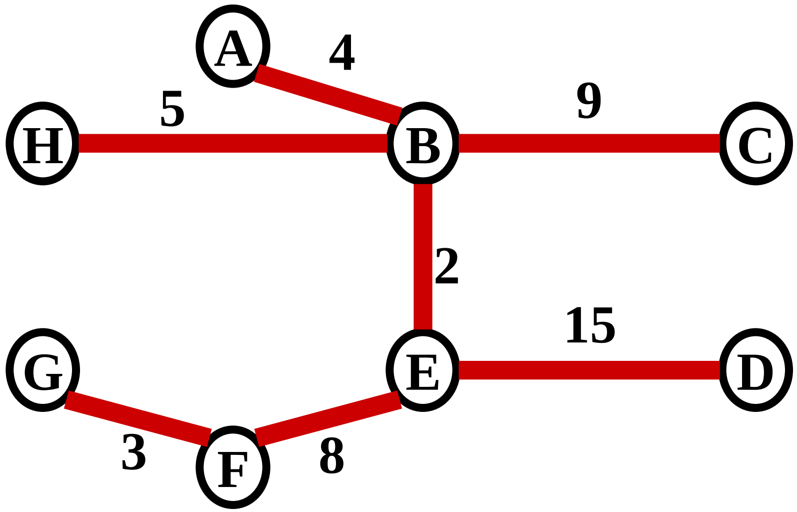
- *Which edges form the minimum spanning tree (MST) of the graph below?*



Minimum Spanning Tree

- Answer:





- Total weight: 3+8+4+5+2+9+15=46. This is smallest total weight.

Two Algorithm

- Krushkal's algorithm
- Prim's algorithm

Kruskal’s Algorithm

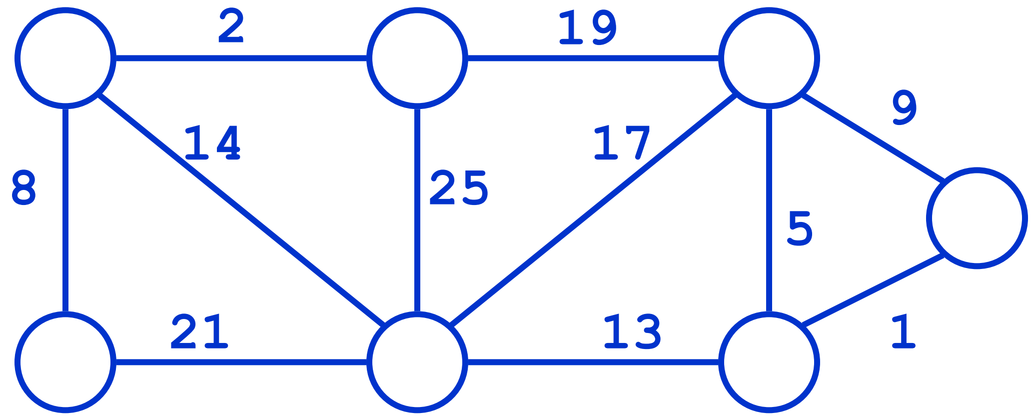
- Take the minimum weight edge one after another if there is no cycle
- Use disjoint set data structure
- MakeSet() for each element
- Maintain two sets:
  - tree edges
  - other remaining edges
- Use FindSet() to check for cycle (If same set, then must be cycle, because if you connect any two vertices of one set, then it makes a cycle.)

Symbols used:
E: Edge set, V: Vertex set, T: Resulting tree, n: number of vertices

Kruskal's Algorithm

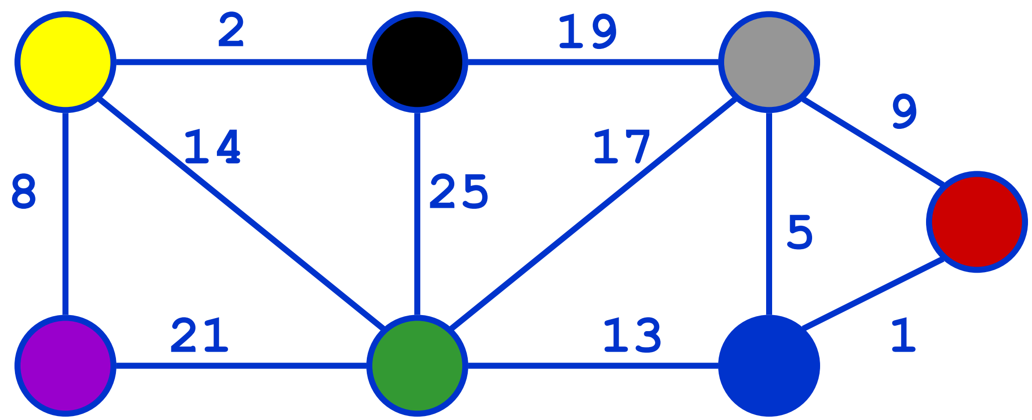
```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v) //no cycle
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

Run the algorithm:



Kruskal's Algorithm

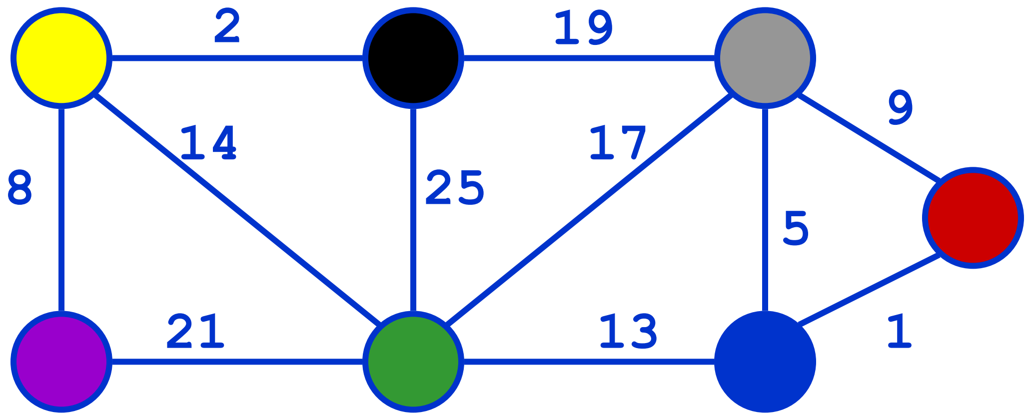
Run the algorithm:




```
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) //no cycle
         T = T U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

Kruskal's Algorithm

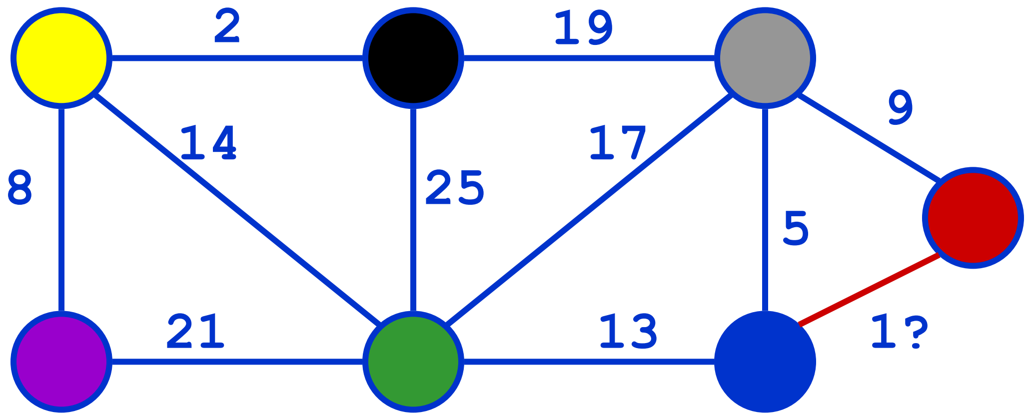
Run the algorithm:




```
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) //no cycle
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

Kruskal's Algorithm

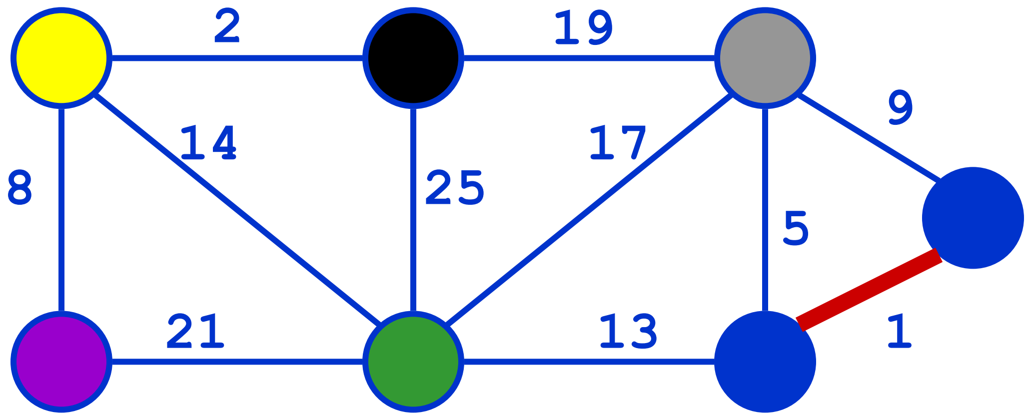
Run the algorithm:




```
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) //no cycle
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

Kruskal's Algorithm

Run the algorithm:

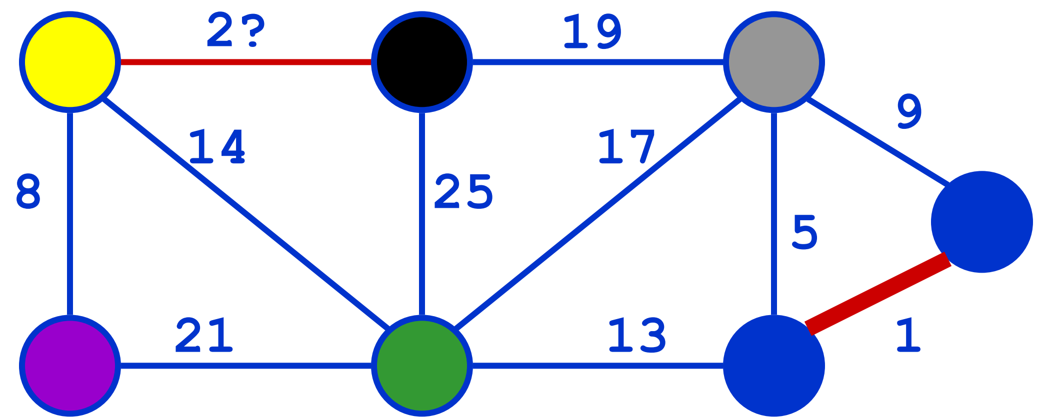



```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v) //no cycle
         T = T U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

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) //no cycle
         T = T U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

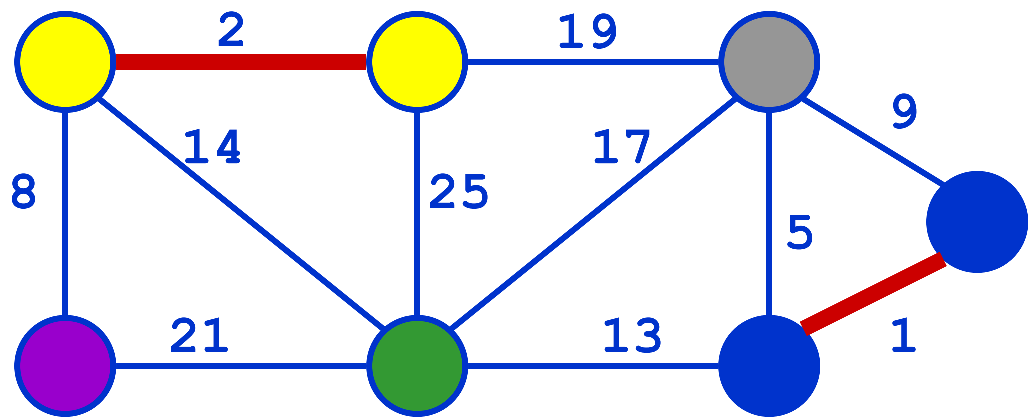
Run the algorithm:



Kruskal's Algorithm

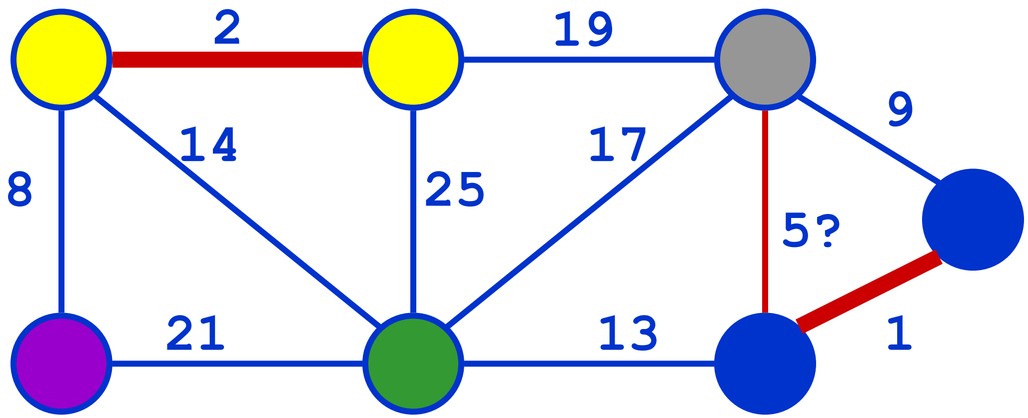
```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v) //no cycle
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

Run the algorithm:



Kruskal's Algorithm

Run the algorithm:

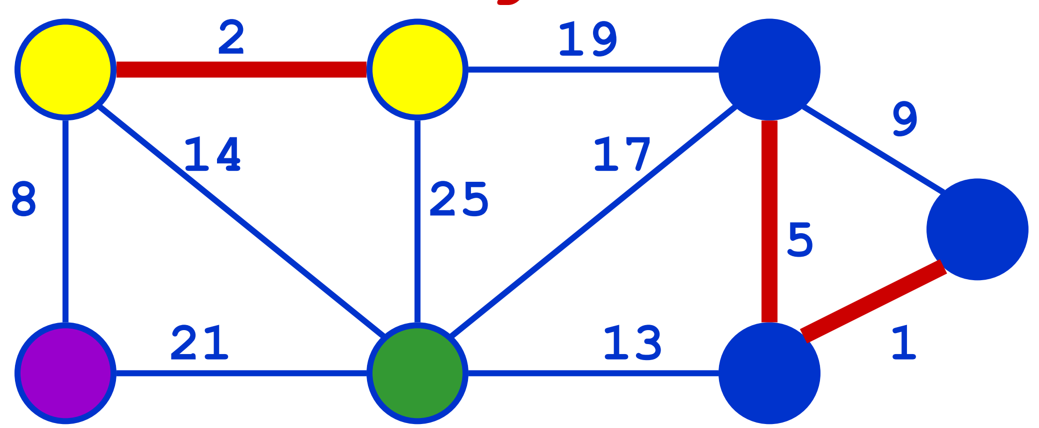



```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v) //no cycle
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

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) //no cycle
         T = T U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

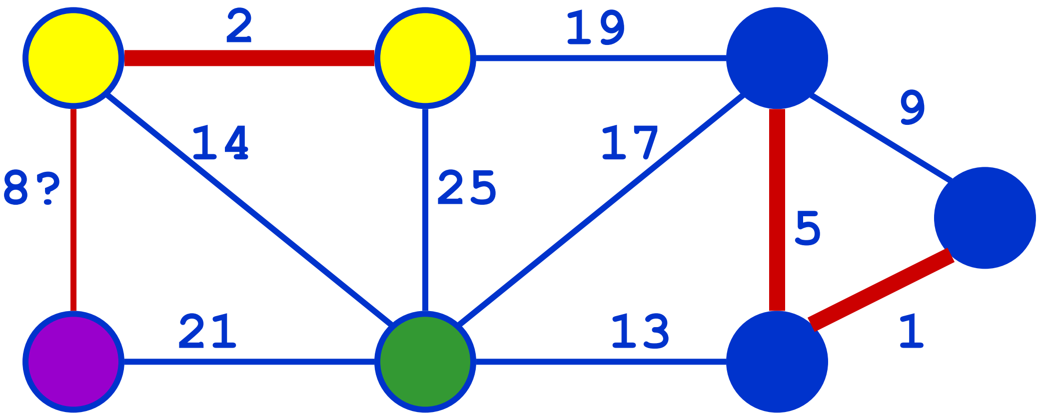
Run the algorithm:



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) //no cycle
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

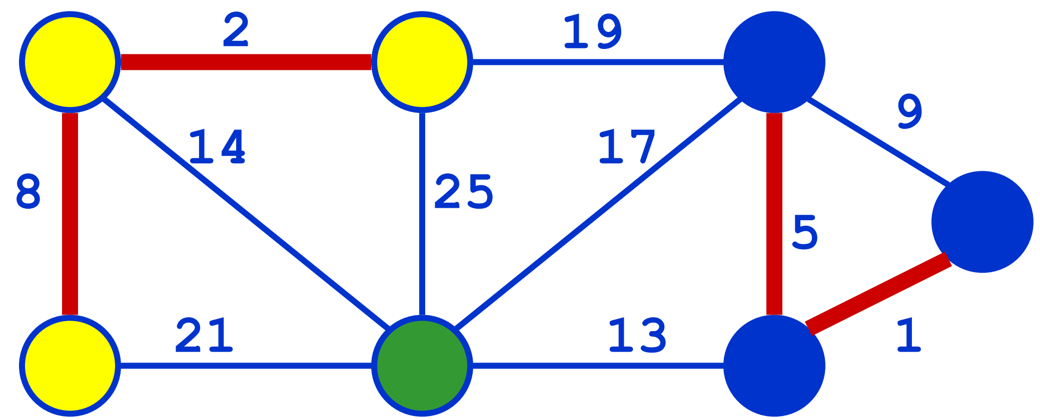
Run the algorithm:



Kruskal's Algorithm

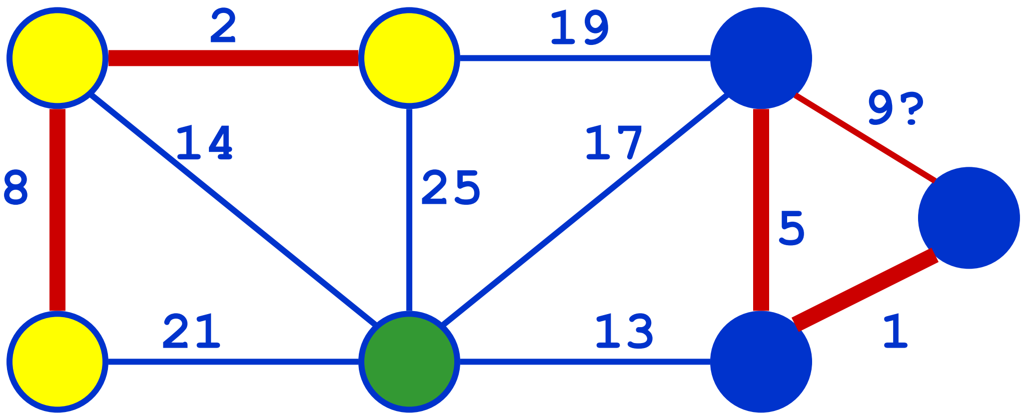
```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v) //no cycle
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

Run the algorithm:



Kruskal's Algorithm

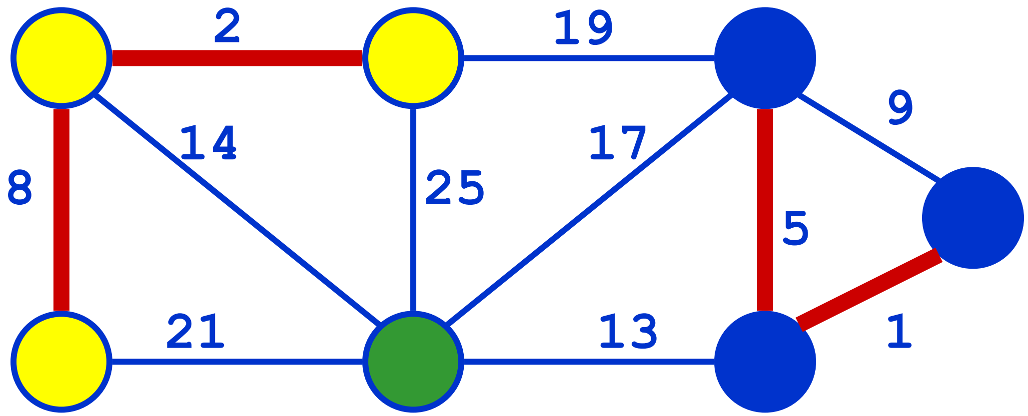
Run the algorithm:




```
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) //no cycle
         T = T U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

Kruskal’s Algorithm

Run the algorithm:

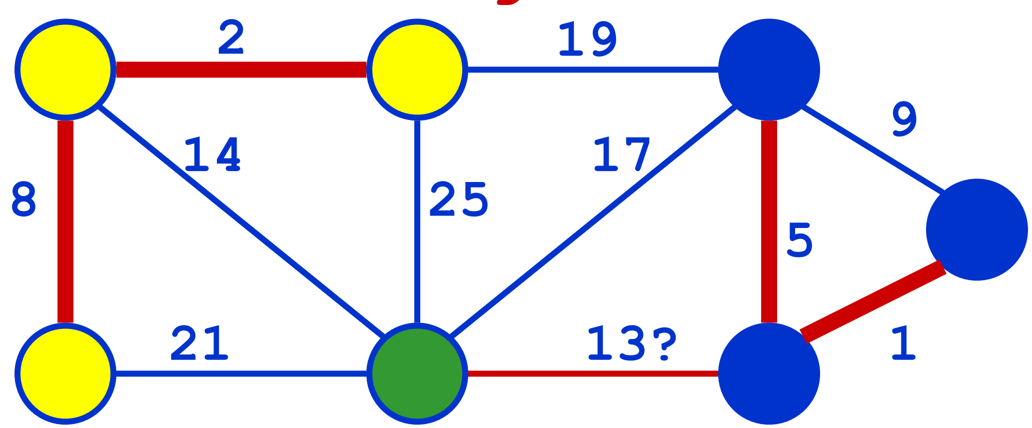



```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v) //no cycle
         T = T U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

Kruskal's Algorithm

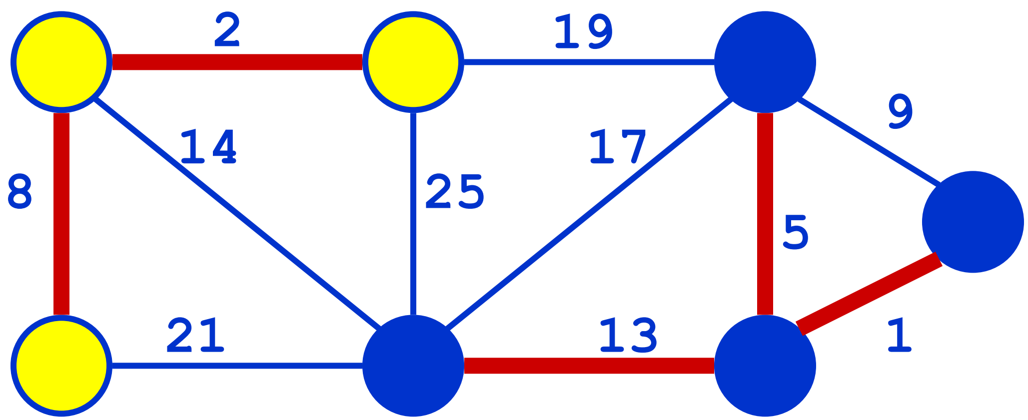
```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v) //no cycle
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

Run the algorithm:



Kruskal's Algorithm

Run the algorithm:

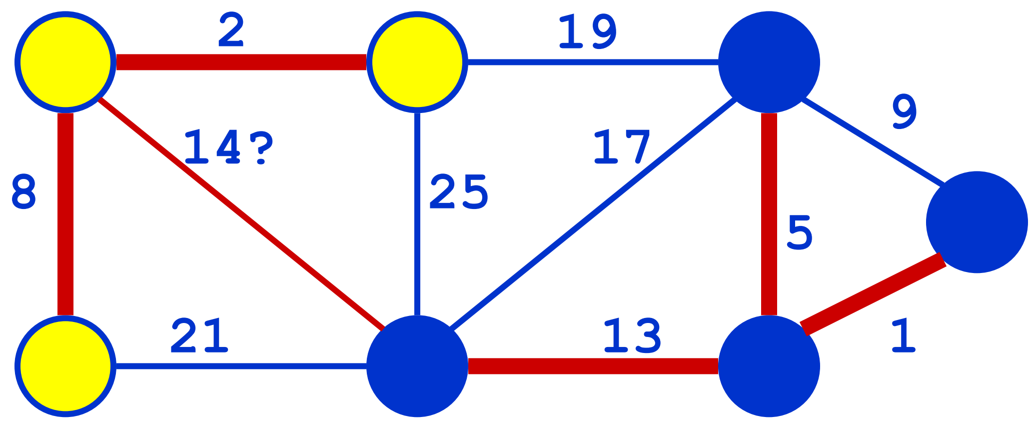



```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v) //no cycle
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

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) //no cycle
         T = T U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

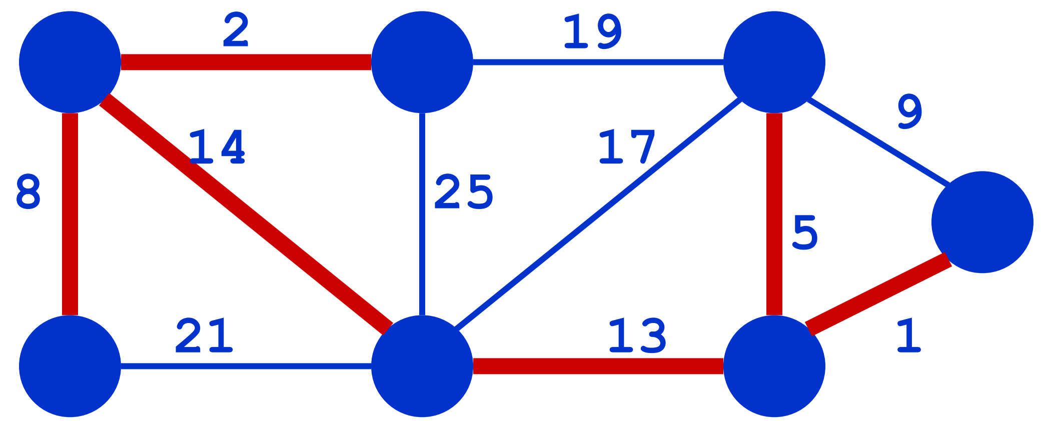
Run the algorithm:



Kruskal's Algorithm

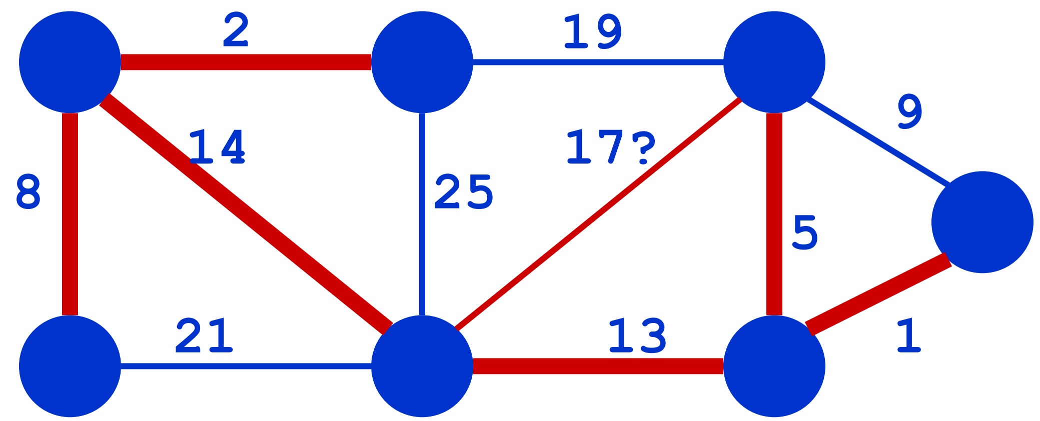
```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v) //no cycle
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

Run the algorithm:



Kruskal's Algorithm

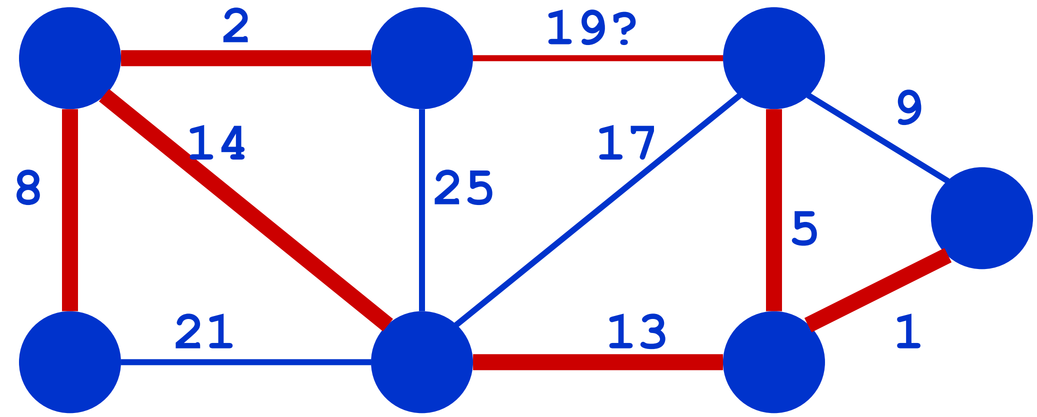
Run the algorithm:




```
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) //no cycle
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

Kruskal's Algorithm

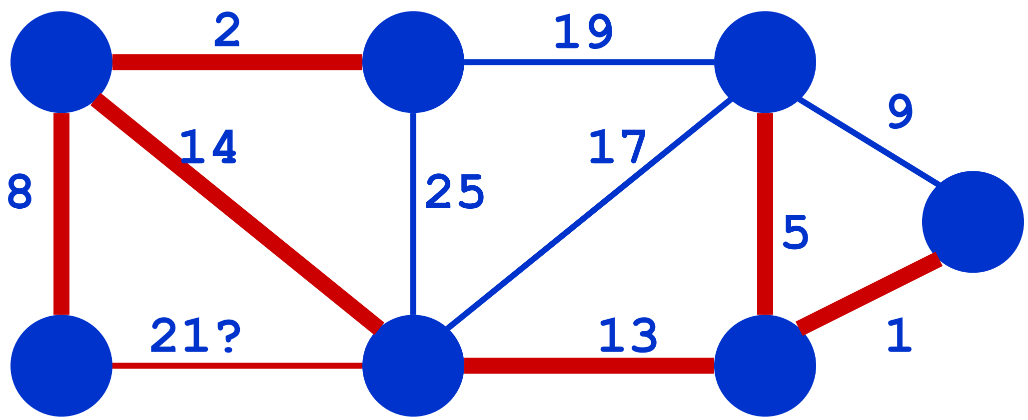
Run the algorithm:




```
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) //no cycle
         T = T U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

Kruskal's Algorithm

Run the algorithm:

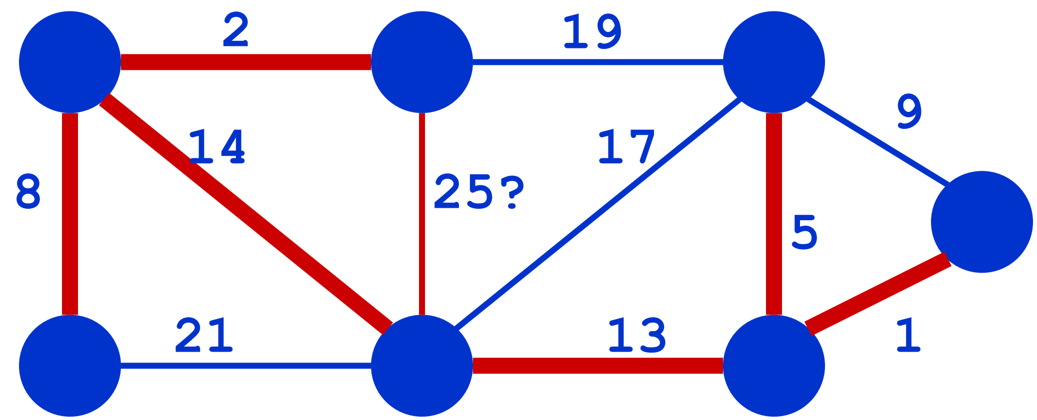



```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v) //no cycle
         T = T U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

Kruskal's Algorithm

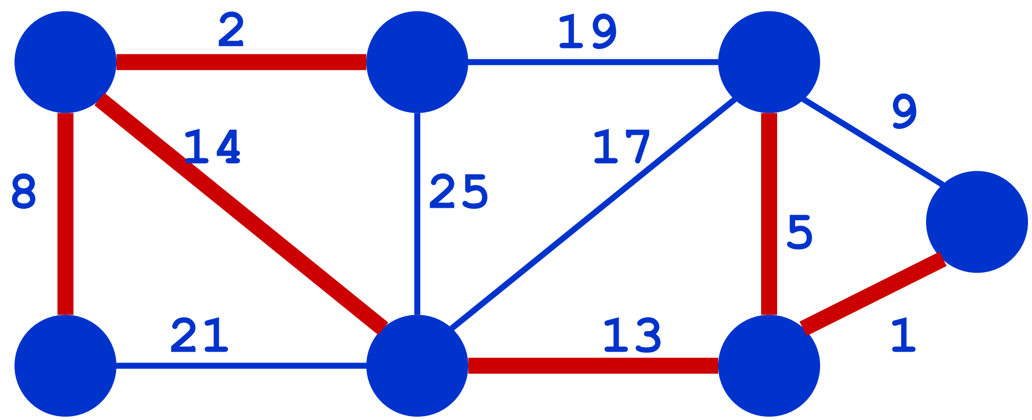
```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v) //no cycle
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

Run the algorithm:



Kruskal's Algorithm

Run the algorithm:

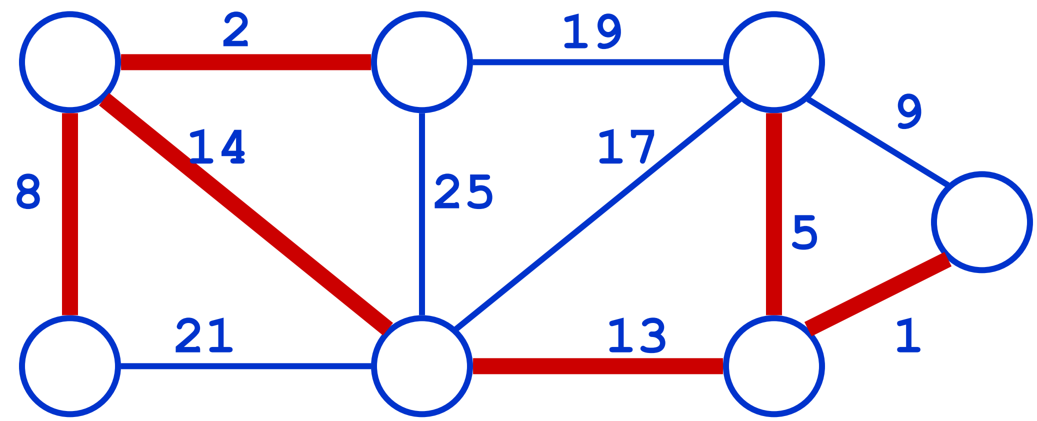



```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v) //no cycle
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

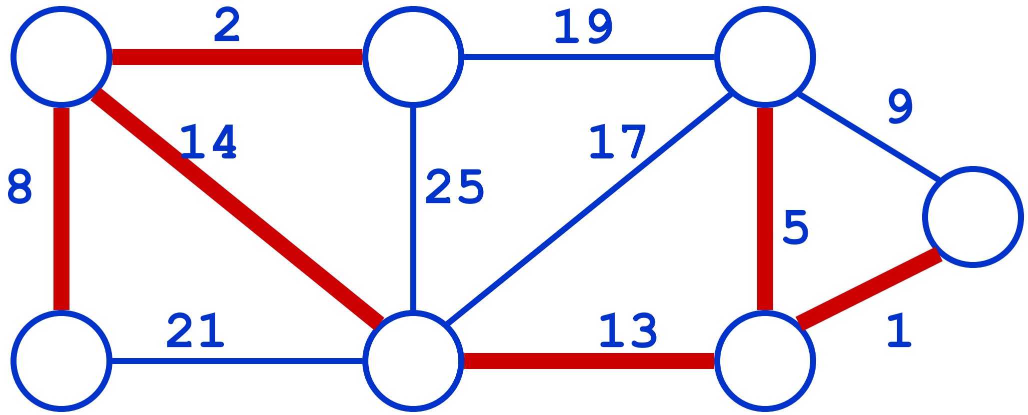
Kruskal's Algorithm

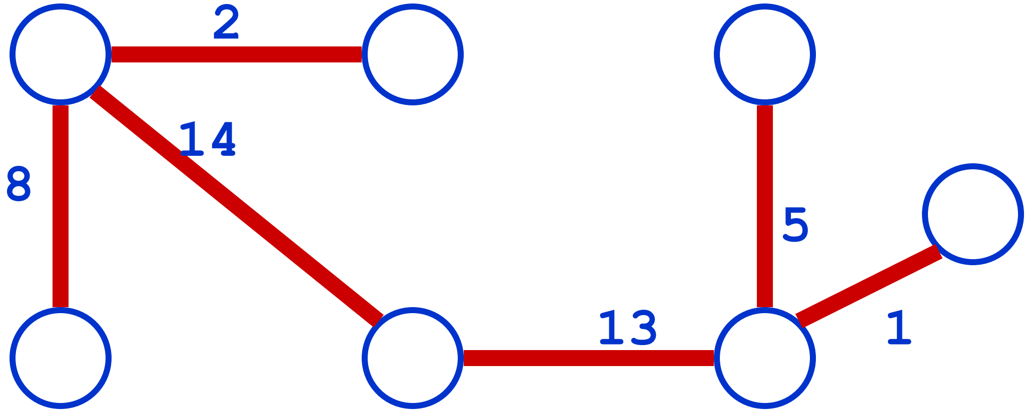
```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v) //no cycle
         T = T U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

Run the algorithm:

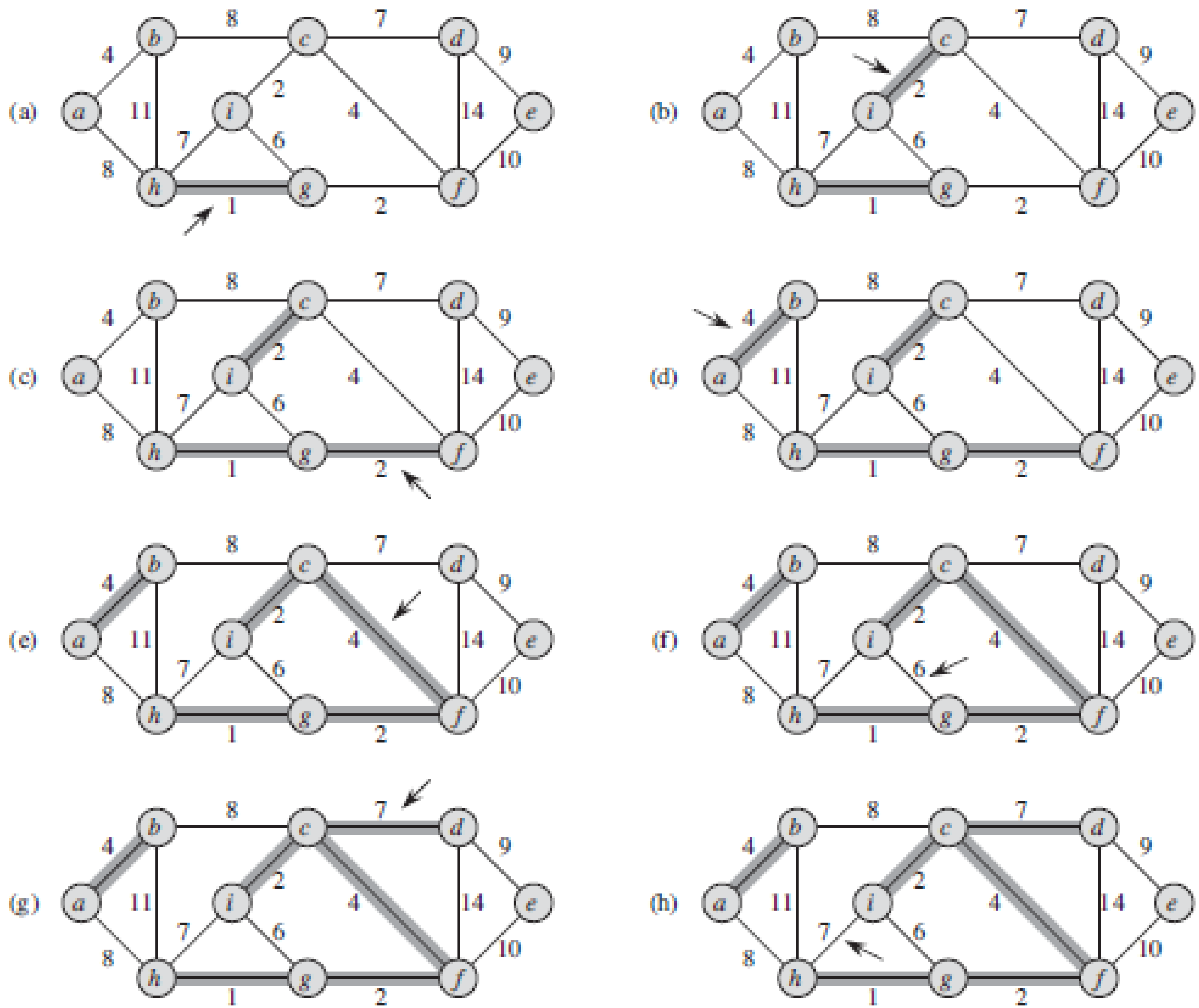


Kruskal's Algorithm

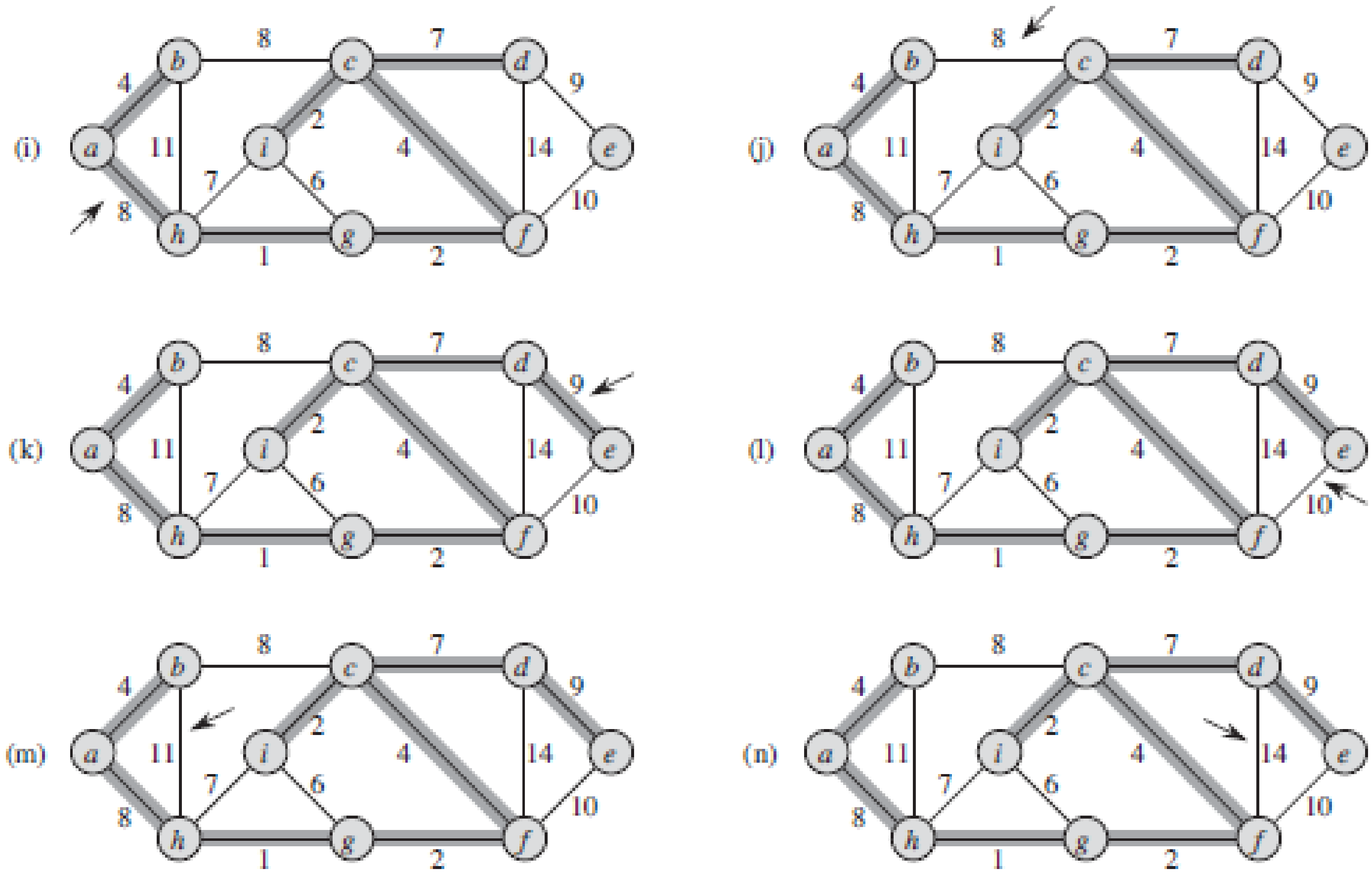




Another Example: Krushkal's Algorithm



Another Example: Krushkal's Algorithm



Cost of 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));
}
```

O(n)

O(E log E)

O(1)

Total E Union(): O(E log E)

Total Cost: O(n) + O(E log E) + O(E log E) = O(E log E)
E: Number of edges, and E > = n

Prim's Algorithm

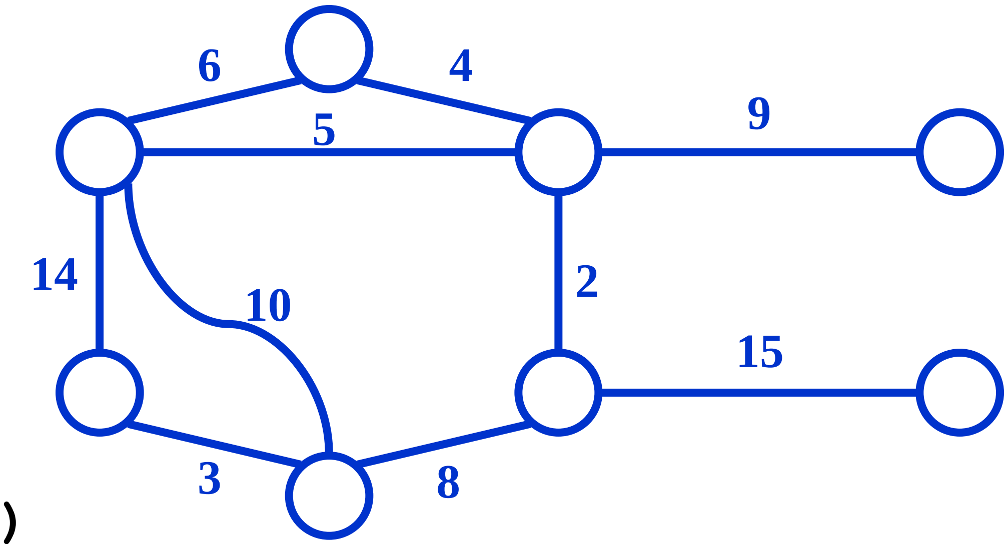
1. Initially, all node have weight ∞
2. Keep all of them in a queue (heap)
3. Take minimum node from heap (ExtractMin())
4. For each neighbor of this node, if edge weight from this node is smaller than current weight , then update weight (DecreaseKey()) and add that edge in the tree.
5. Repeat step 3, 4 until finished.

n: number of vertex

Prim's Algorithm

Q: heap, V: vertex set, E: Edge set, p: parent, w: edge weight

```
MST-Prim
    for all node u
        weight[u] = ∞;
    Q = all node;
    weight[root] = 0;
    p[root] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adjacent[u]
            if (v ∈ Q and w(u,v) < weight[v])
                p[v] = u;
                weight[v] = w(u,v);
```

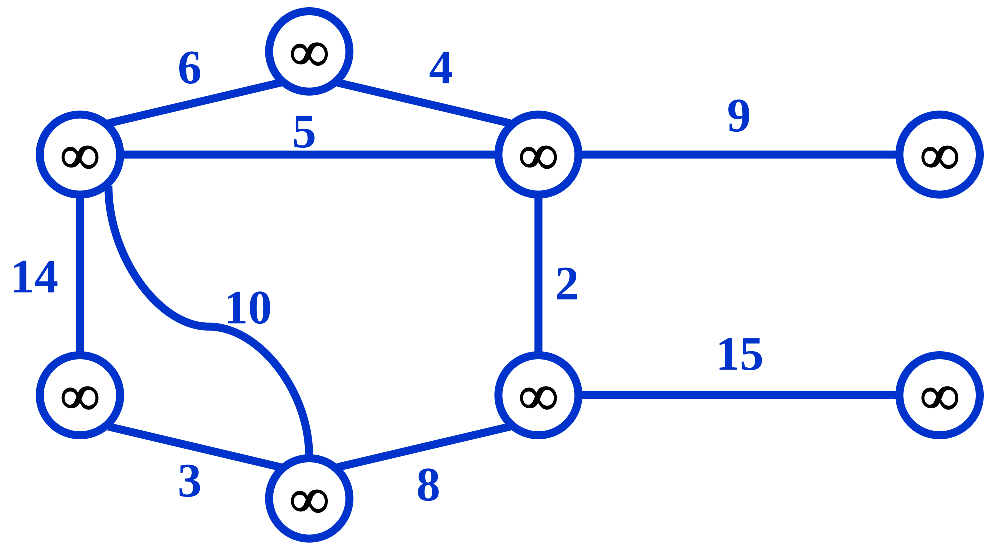


Run on example graph

Prim's Algorithm

Q: heap, V: vertex set, E: Edge set, p: parent, w: edge weight

```
MST-Prim
    for all node u
        weight[u] = ∞;
    Q = all node;
    weight[root] = 0;
    p[root] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adjacent[u]
            if (v ∈ Q and w(u,v) < weight[v])
                p[v] = u;
                weight[v] = w(u,v);
```

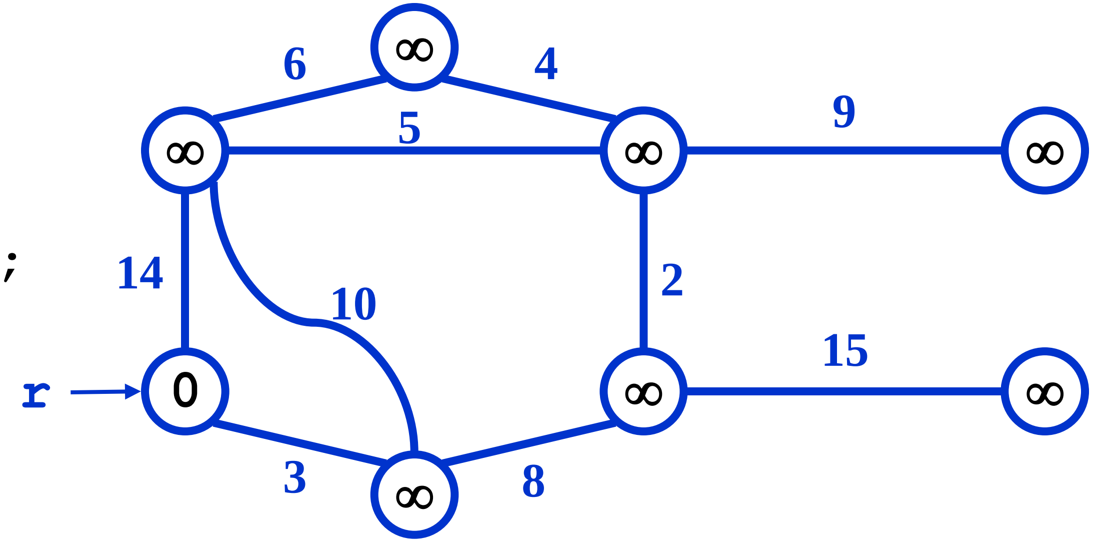


Run on example graph

Prim's Algorithm

Q: heap, V: vertex set, E: Edge set, p: parent, w: edge weight

```
MST-Prim
    for all node u
        weight[u] = ∞;
    Q = all node;
    weight[root] = 0;
    p[root] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adjacent[u]
            if (v ∈ Q and w(u,v) < weight[v])
                p[v] = u;
                weight[v] = w(u,v);
```

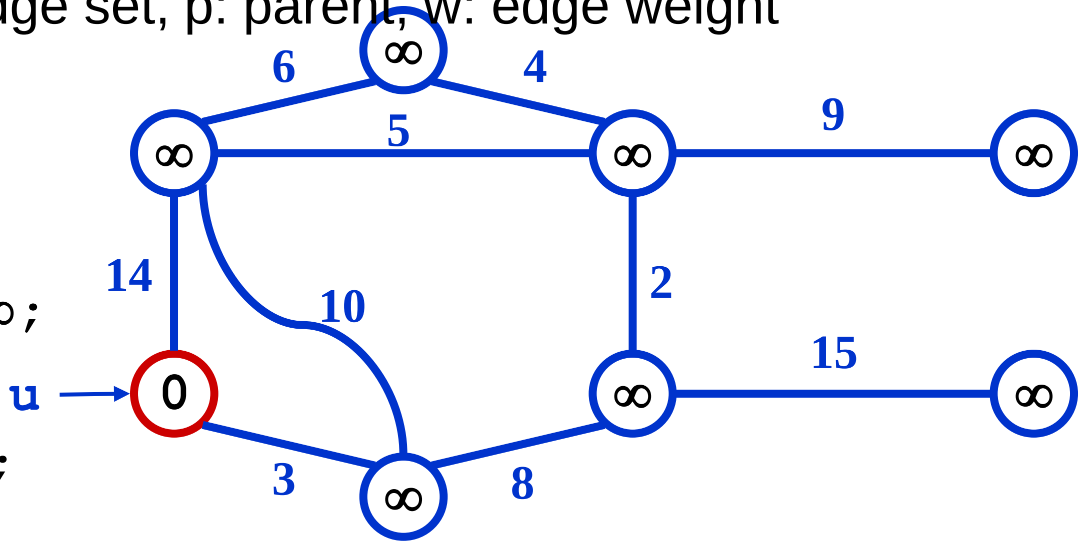


Pick a start vertex r

Prim’s Algorithm

Q: heap, V: vertex set, E: Edge set, p: parent, w: edge weight

```
MST-Prim
    for all node u
        weight[u] = ∞;
    Q = all node;
    weight[root] = 0;
    p[root] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adjacent[u]
            if (v ∈ Q and w(u,v) < weight[v])
                p[v] = u;
                weight[v] = w(u,v);
```

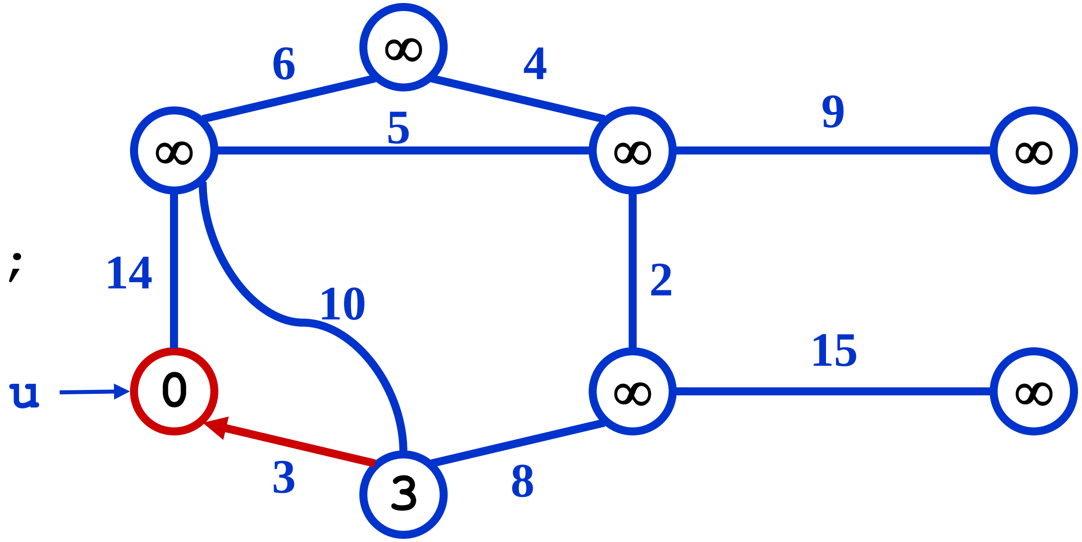


Red vertices have been removed from Q

Prim's Algorithm

Q: heap, V: vertex set, E: Edge set, p: parent, w: edge weight

```
MST-Prim
    for all node u
        weight[u] = ∞;
    Q = all node;
    weight[root] = 0;
    p[root] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adjacent[u]
            if (v ∈ Q and w(u,v) < weight[v])
                p[v] = u;
                weight[v] = w(u,v);
```

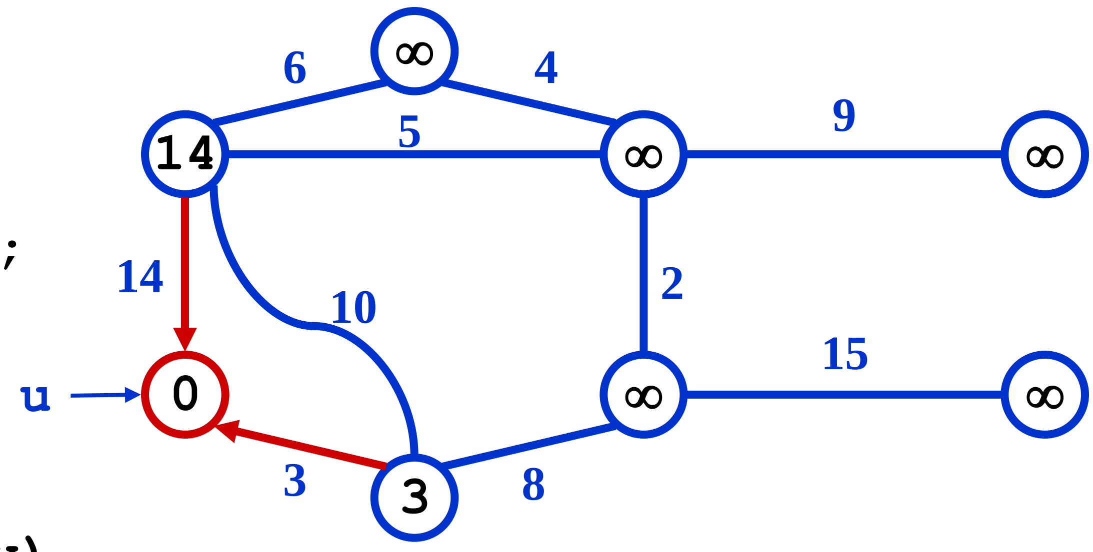


Red arrows indicate parent pointers

Prim's Algorithm

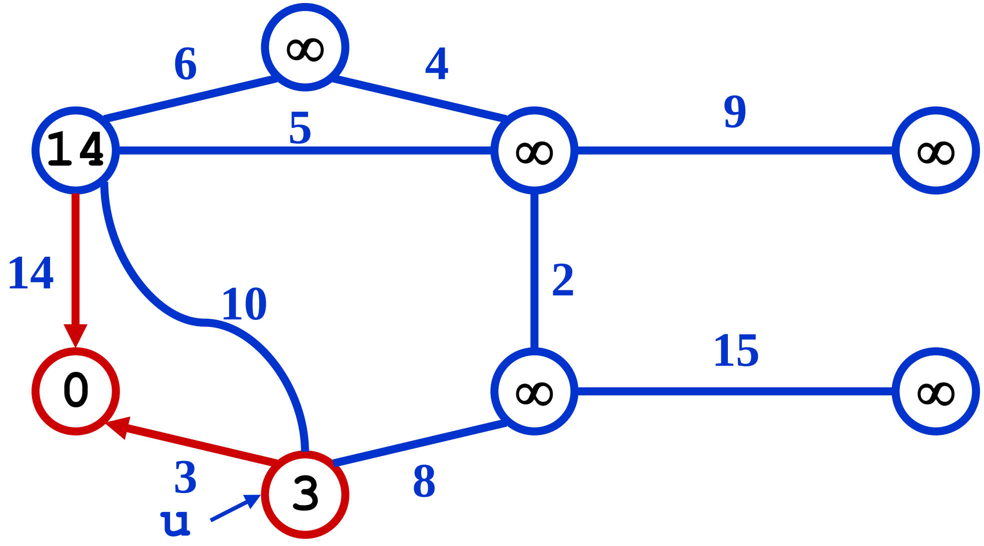
Q: heap, V: vertex set, E: Edge set, p: parent, w: edge weight

```
MST-Prim
    for all node u
        weight[u] = ∞;
    Q = all node;
    weight[root] = 0;
    p[root] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adjacent[u]
            if (v ∈ Q and w(u,v) < weight[v])
                p[v] = u;
                weight[v] = w(u,v);
```



Prim's Algorithm

Q: heap, V: vertex set, E: Edge set, p: parent, w: edge weight

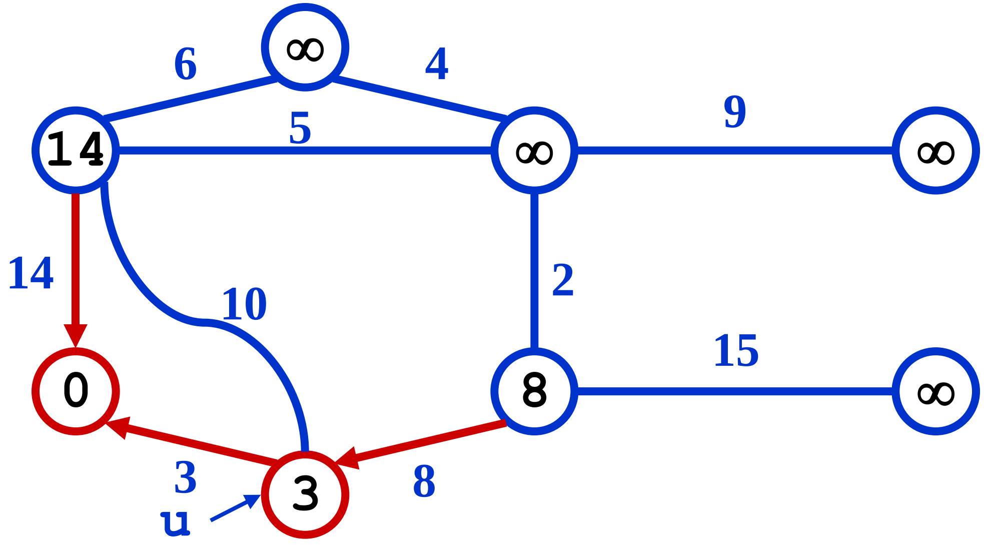



```
MST-Prim
    for all node u
        weight[u] = ∞;
    Q = all node;
    weight[root] = 0;
    p[root] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adjacent[u]
            if (v ∈ Q and w(u,v) < weight[v])
                p[v] = u;
                weight[v] = w(u,v);
```

Prim's Algorithm

Q: heap, V: vertex set, E: Edge set, p: parent, w: edge weight

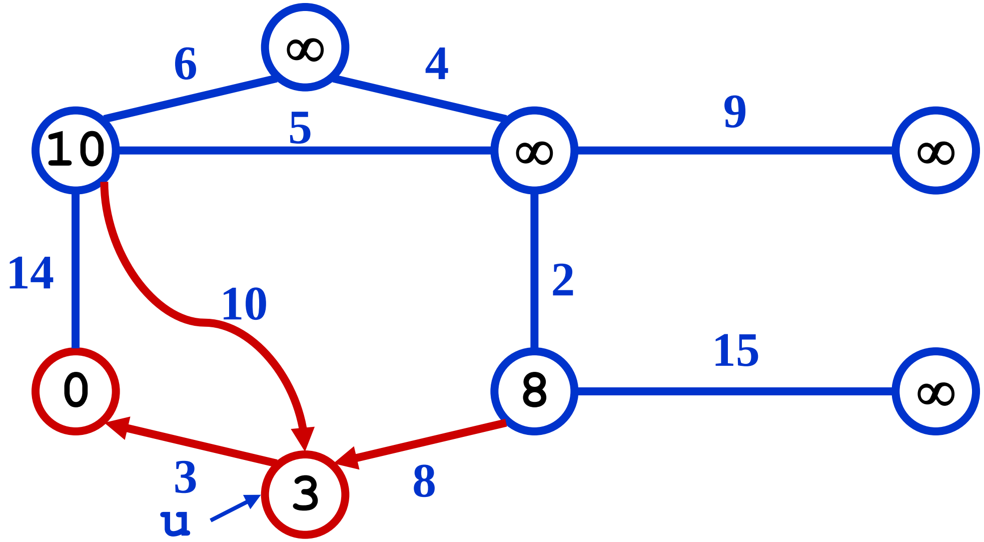
```
MST-Prim
    for all node u
        weight[u] = ∞;
    Q = all node;
    weight[root] = 0;
    p[root] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adjacent[u]
            if (v ∈ Q and w(u,v) < weight[v])
                p[v] = u;
                weight[v] = w(u,v);
```



Prim's Algorithm

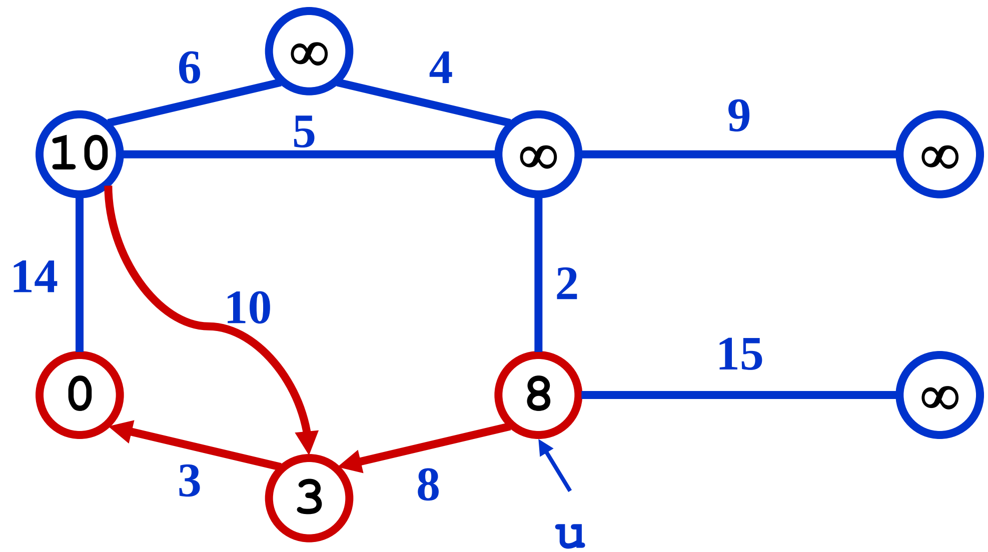
Q: heap, V: vertex set, E: Edge set, p: parent, w: edge weight

```
MST-Prim
    for all node u
        weight[u] = ∞;
    Q = all node;
    weight[root] = 0;
    p[root] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adjacent[u]
            if (v ∈ Q and w(u,v) < weight[v])
                p[v] = u;
                weight[v] = w(u,v);
```



Prim's Algorithm

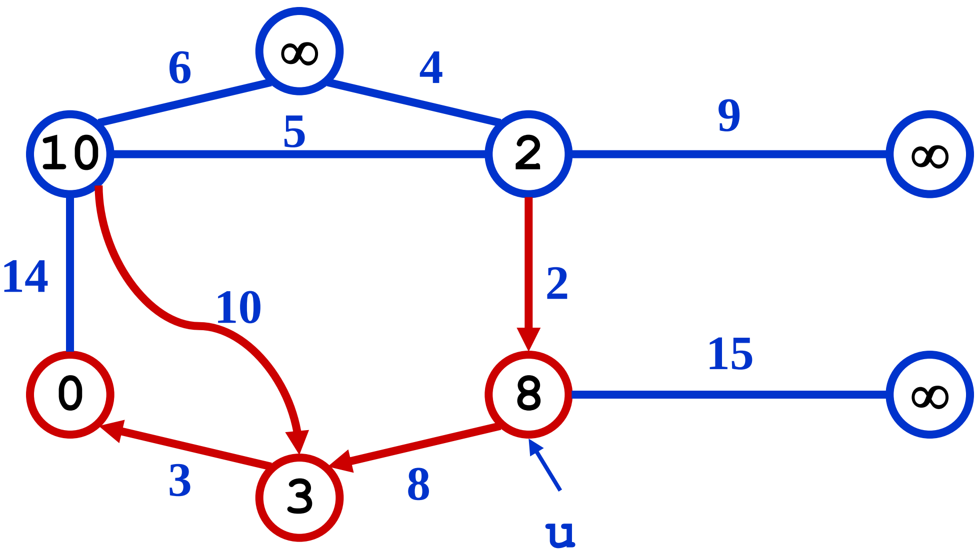
Q: heap, V: vertex set, E: Edge set, p: parent, w: edge weight




```
MST-Prim
    for all node u
        weight[u] = ∞;
    Q = all node;
    weight[root] = 0;
    p[root] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adjacent[u]
            if (v ∈ Q and w(u,v) < weight[v])
                p[v] = u;
                weight[v] = w(u,v);
```

Prim's Algorithm

Q: heap, V: vertex set, E: Edge set, p: parent, w: edge weight

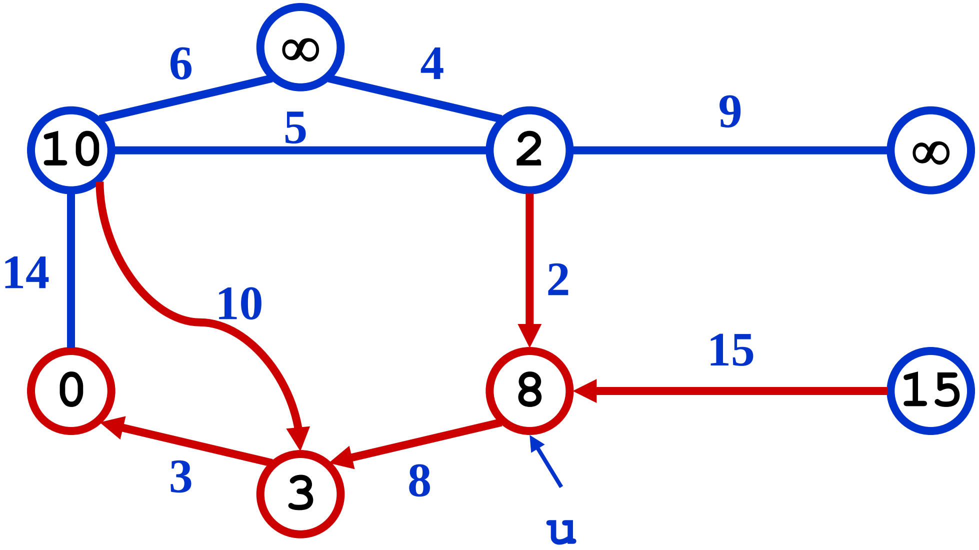



```
MST-Prim
    for all node u
        weight[u] = ∞;
    Q = all node;
    weight[root] = 0;
    p[root] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adjacent[u]
            if (v ∈ Q and w(u,v) < weight[v])
                p[v] = u;
                weight[v] = w(u,v);
```

Prim's Algorithm

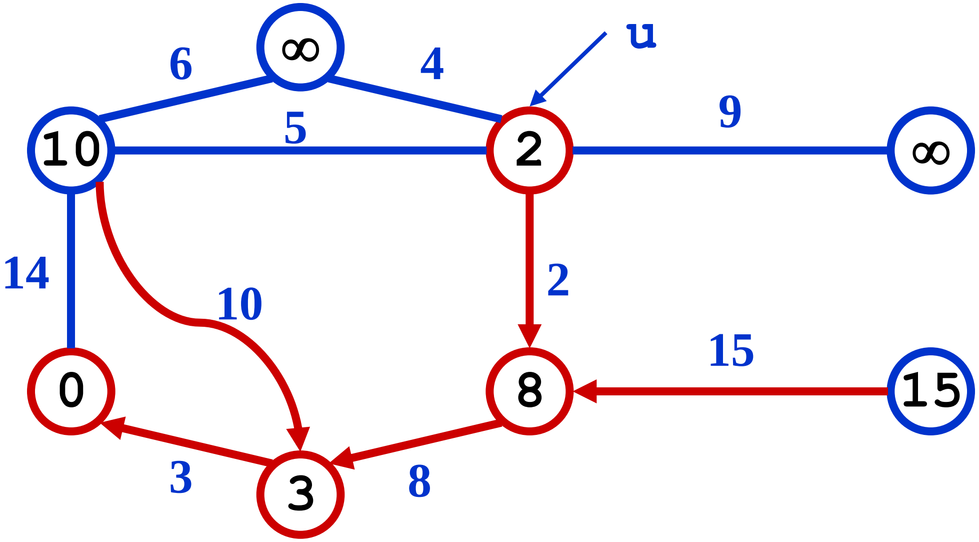
Q: heap, V: vertex set, E: Edge set, p: parent, w: edge weight

```
MST-Prim
    for all node u
        weight[u] = ∞;
    Q = all node;
    weight[root] = 0;
    p[root] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adjacent[u]
            if (v ∈ Q and w(u,v) < weight[v])
                p[v] = u;
                weight[v] = w(u,v);
```



Prim's Algorithm

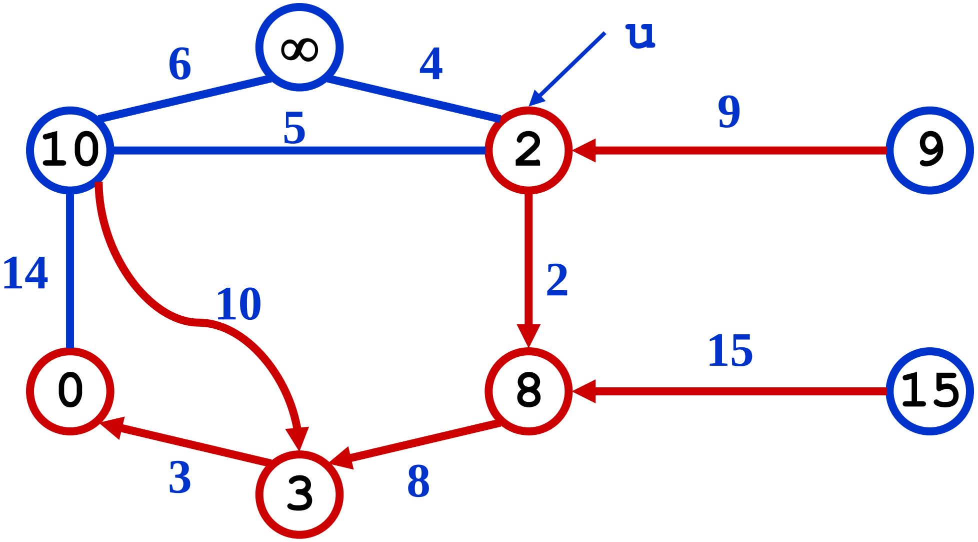
Q: heap, V: vertex set, E: Edge set, p: parent, w: edge weight




```
MST-Prim
    for all node u
        weight[u] = ∞;
    Q = all node;
    weight[root] = 0;
    p[root] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adjacent[u]
            if (v ∈ Q and w(u,v) < weight[v])
                p[v] = u;
                weight[v] = w(u,v);
```

Prim's Algorithm

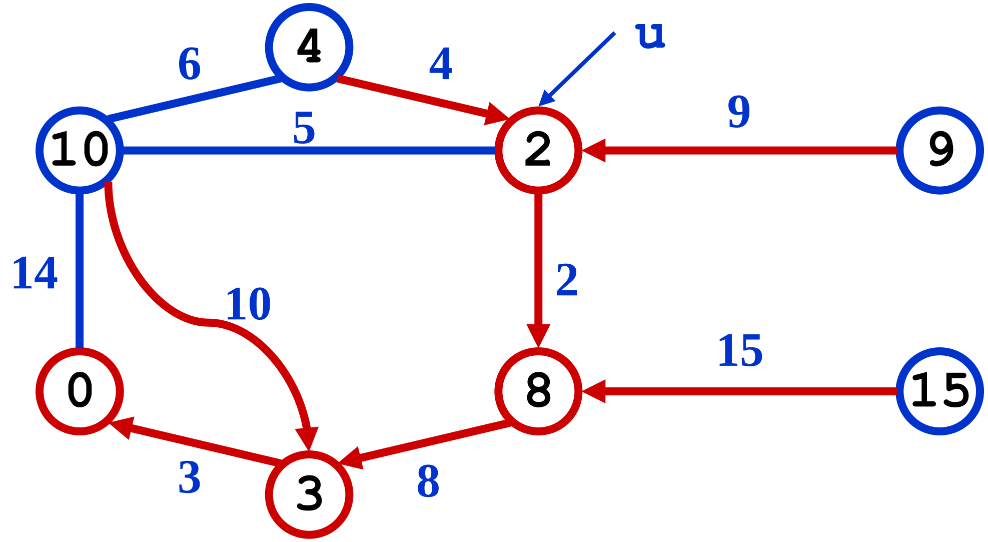
Q: heap, V: vertex set, E: Edge set, p: parent, w: edge weight




```
MST-Prim
    for all node u
        weight[u] = ∞;
    Q = all node;
    weight[root] = 0;
    p[root] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adjacent[u]
            if (v ∈ Q and w(u,v) < weight[v])
                p[v] = u;
                weight[v] = w(u,v);
```

Prim's Algorithm

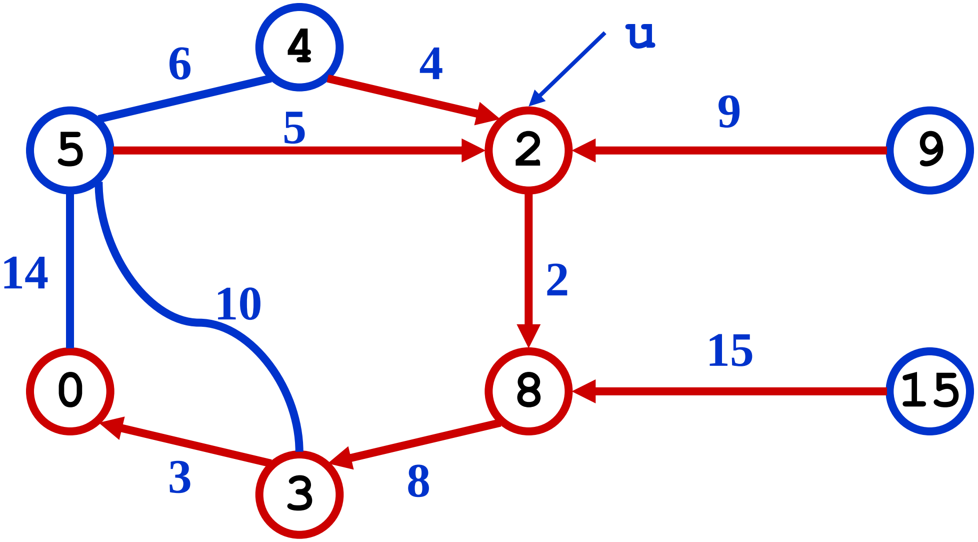
Q: heap, V: vertex set, E: Edge set, p: parent, w: edge weight




```
MST-Prim
    for all node u
        weight[u] = ∞;
    Q = all node;
    weight[root] = 0;
    p[root] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adjacent[u]
            if (v ∈ Q and w(u,v) < weight[v])
                p[v] = u;
                weight[v] = w(u,v);
```

Prim's Algorithm

Q: heap, V: vertex set, E: Edge set, p: parent, w: edge weight

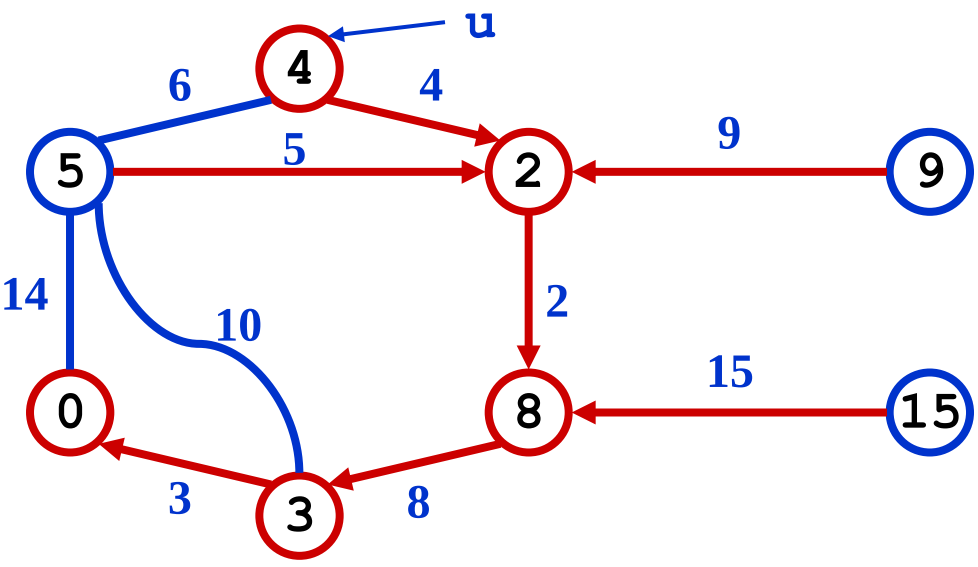



```
MST-Prim
    for all node u
        weight[u] = ∞;
    Q = all node;
    weight[root] = 0;
    p[root] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adjacent[u]
            if (v ∈ Q and w(u,v) < weight[v])
                p[v] = u;
                weight[v] = w(u,v);
```

Prim's Algorithm

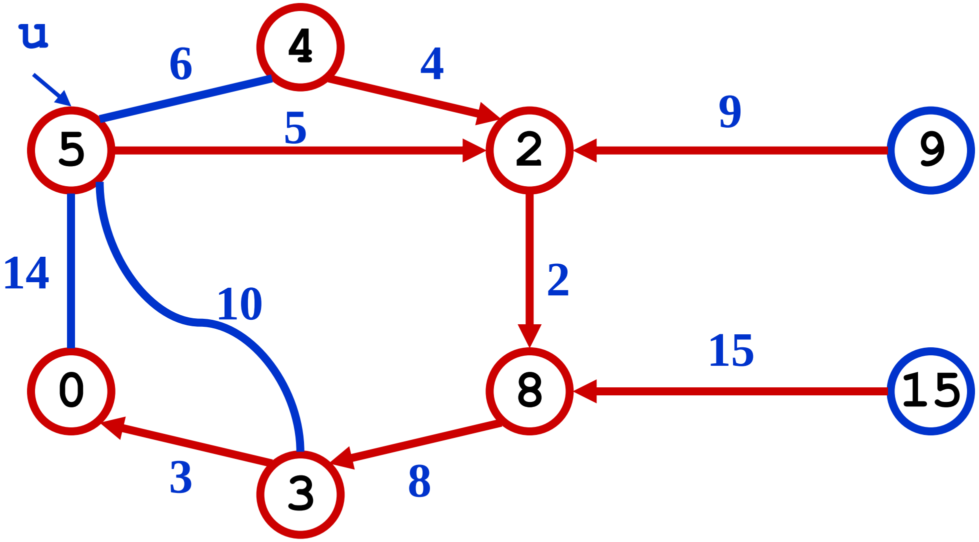
Q: heap, V: vertex set, E: Edge set, p: parent, w: edge weight

```
MST-Prim
    for all node u
        weight[u] = ∞;
    Q = all node;
    weight[root] = 0;
    p[root] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adjacent[u]
            if (v ∈ Q and w(u,v) < weight[v])
                p[v] = u;
                weight[v] = w(u,v);
```



Prim's Algorithm

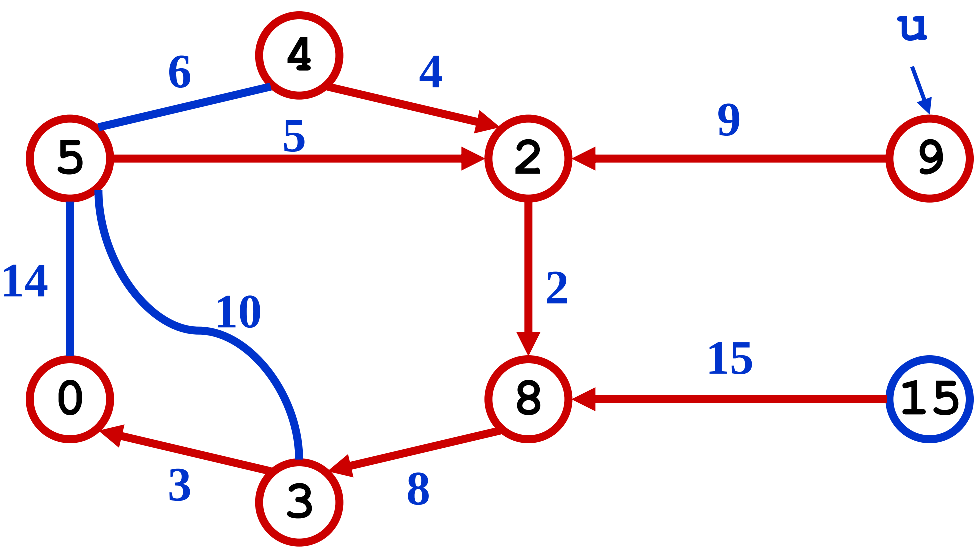
Q: heap, V: vertex set, E: Edge set, p: parent, w: edge weight




```
MST-Prim
    for all node u
        weight[u] = ∞;
    Q = all node;
    weight[root] = 0;
    p[root] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adjacent[u]
            if (v ∈ Q and w(u,v) < weight[v])
                p[v] = u;
                weight[v] = w(u,v);
```

Prim's Algorithm

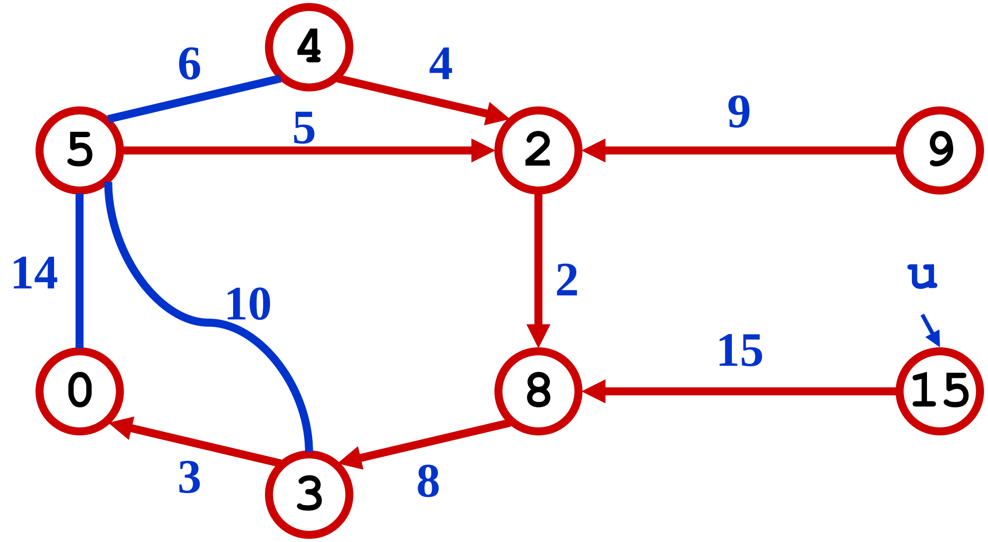
Q: heap, V: vertex set, E: Edge set, p: parent, w: edge weight




```
MST-Prim
     for all node u
          weight[u] = ∞;
     Q = all node;
     weight[root] = 0;
     p[root] = NULL;
     while (Q not empty)
          u = ExtractMin(Q);
          for each v ∈ Adjacent[u]
               if (v ∈ Q and w(u,v) < weight[v])
                    p[v] = u;
                    weight[v] = w(u,v);
```

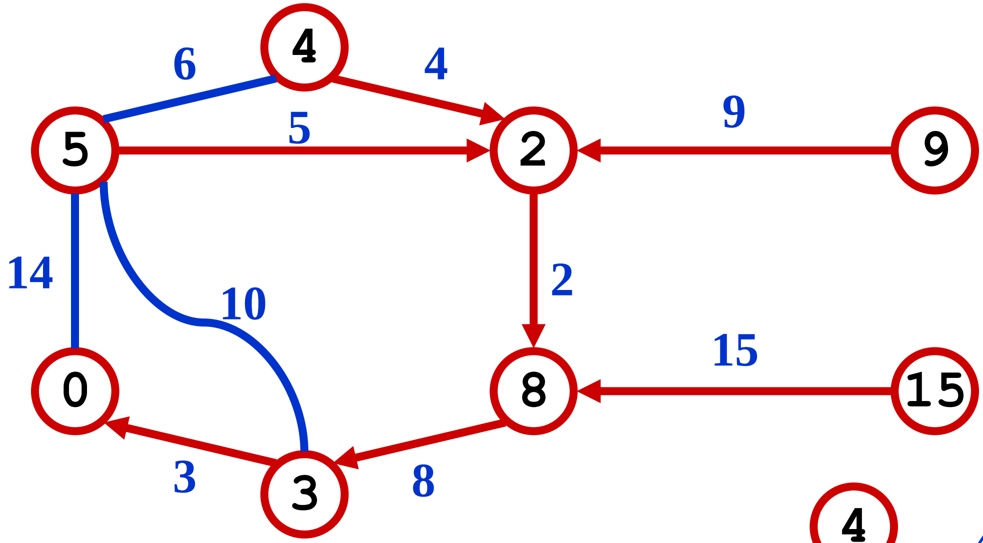
Prim’s Algorithm

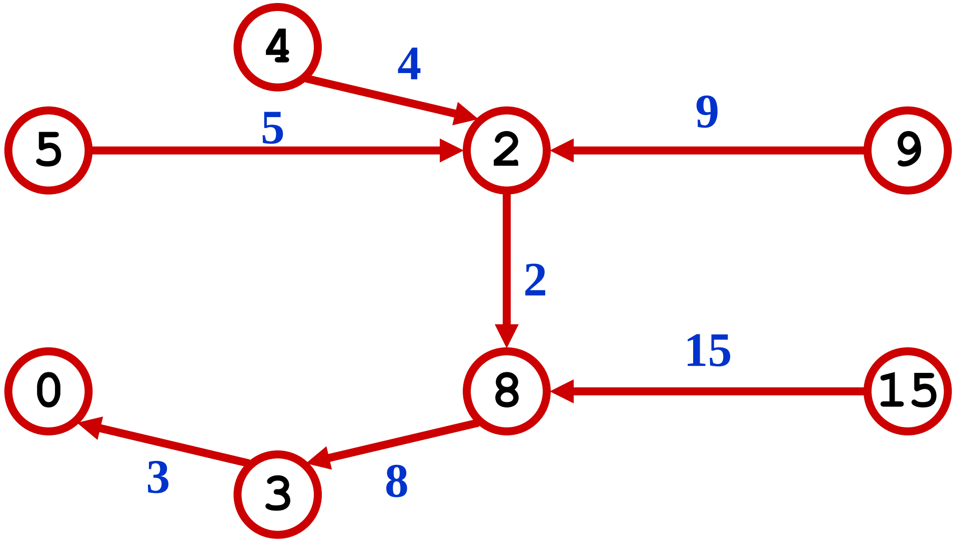
Q: heap, V: vertex set, E: Edge set, p: parent, w: edge weight




```
MST-Prim
    for all node u
        weight[u] = ∞;
    Q = all node;
    weight[root] = 0;
    p[root] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adjacent[u]
            if (v ∈ Q and w(u,v) < weight[v])
                p[v] = u;
                weight[v] = w(u,v);
```

Prim's Algorithm





Cost of Prim's Algorithm

```
MST-Prim
    for each node u
        weight[u] = ∞;
    Q = all node;
    weight[root] = 0;
    p[root] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adjacent[u]
            if (v ∈ Q and w(u,v) < weight[v])
                p[v] = u;
                weight[v] = w(u,v);
```

O(n)

BuildHeap = O(n log n)

n times

HeapExtractMin = O(log n)

Total for all nodes = E times

HeapDecreaseKey = O(log n)

Worst Case Total for this part: n*(O(log n)+E*O(log n)) = O(n log n) + O(nE log n) = O(nE log n)

Cost of Prim's Algorithm

```
MST-Prim
    for each node u                                  O(n)
        weight[u] = ∞;
    Q = all node;                                    BuildHeap = O(n log n)
    weight[root] = 0;
    p[root] = NULL;
    while (Q not empty)                              n times
        u = ExtractMin(Q);                           HeapExtractMin = O(log n)
        for each v ∈ Adjacent[u]                     Total for all nodes =
            if (v ∈ Q and w(u,v) < weight[v])            E times
                p[v] = u;
                weight[v] = w(u,v);                  HeapDecreaseKey
                                                     = O(log n)
```

But, amortized cost (agreegate method): How many total ExractMin()? **Answer:** n. How many total DecreaseKey()? **Answer:** E.

Cost of Prim's Algorithm

```
MST-Prim
    for each node u                              ← O(n)
        weight[u] = ∞;
    Q = all node;                                ← BuildHeap = O(n log n)
    weight[root] = 0;
    p[root] = NULL;
    while (Q not empty)                          ← n times
        u = ExtractMin(Q);                       ← HeapExtractMin = O(log n)
        for each v ∈ Adjacent[u]                 ← Total for all nodes = E times
            if (v ∈ Q and w(u,v) < weight[v])
                p[v] = u;
                weight[v] = w(u,v);              ← HeapDecreaseKey = O(log n)
```

So, total cost: BuildHeap() + n ExtractMin + E DecreaseKey = n*O(log n)+ E*O(n log n) = O(n og n) + O(E log n) = **O(E log n),** because E >= n