Graph Algorithms

Part 2 MST

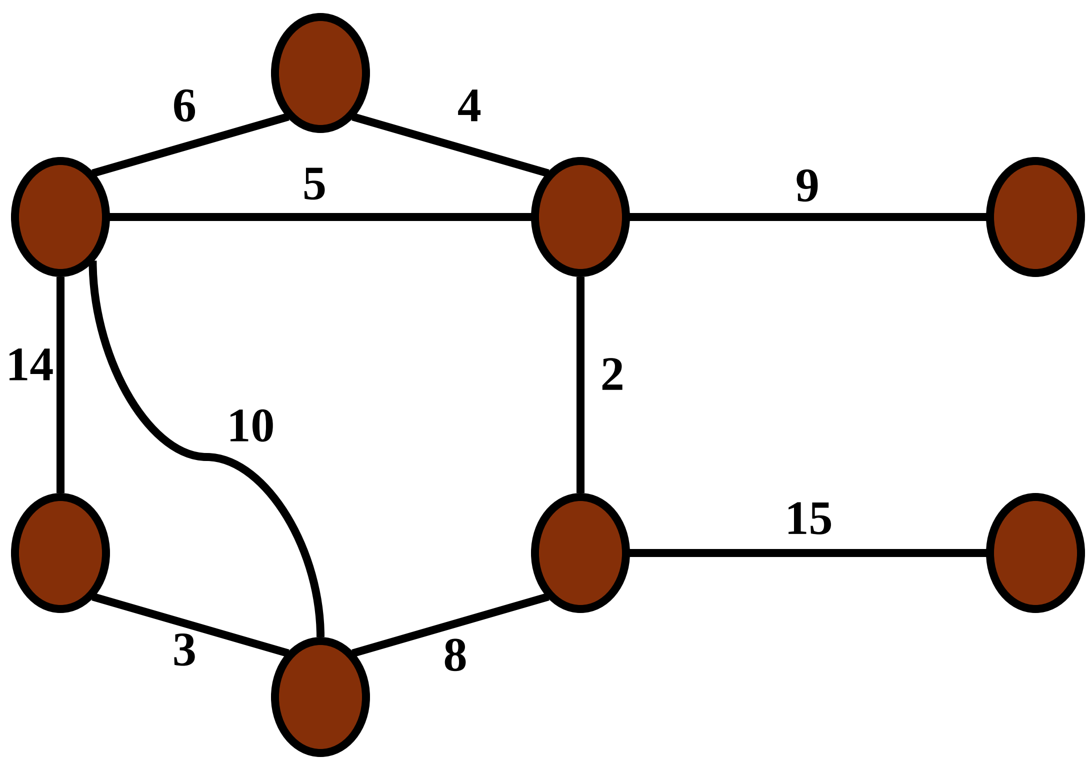
CS 325

Ch 23 Spanning Trees

- Weighted Graphs
- Minimum Spanning Trees
  - Greedy Choice Theorem
  - Kruskal's Algorithm
  - Prim's Algorithm

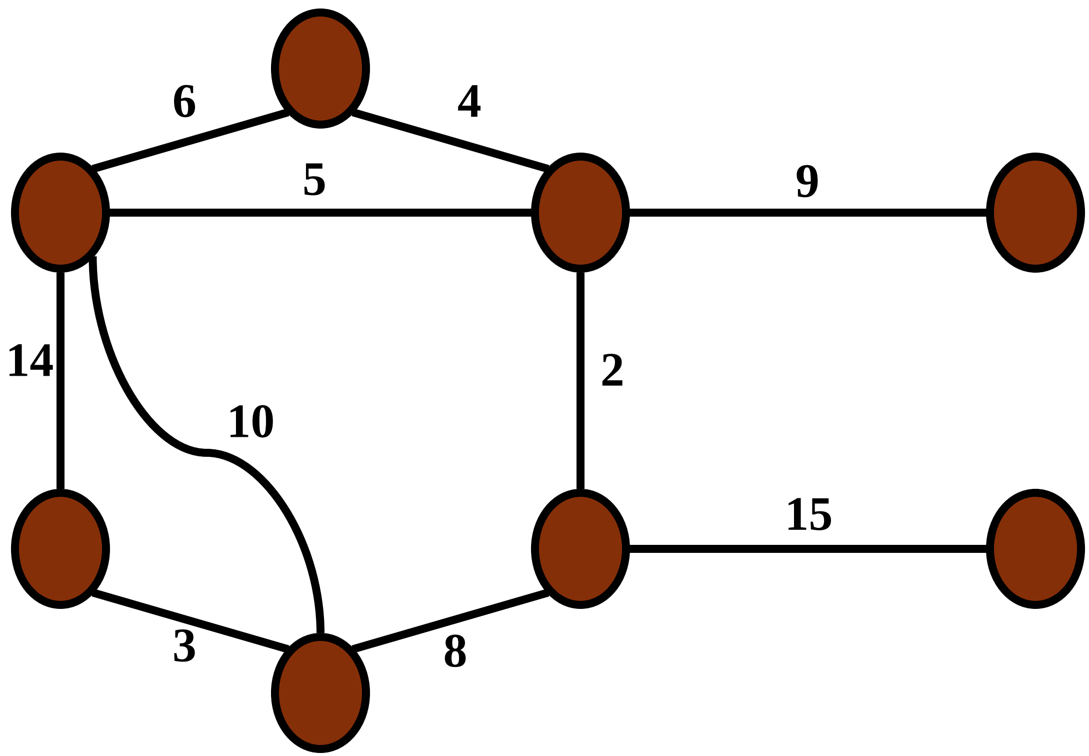
Minimum Spanning Tree

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



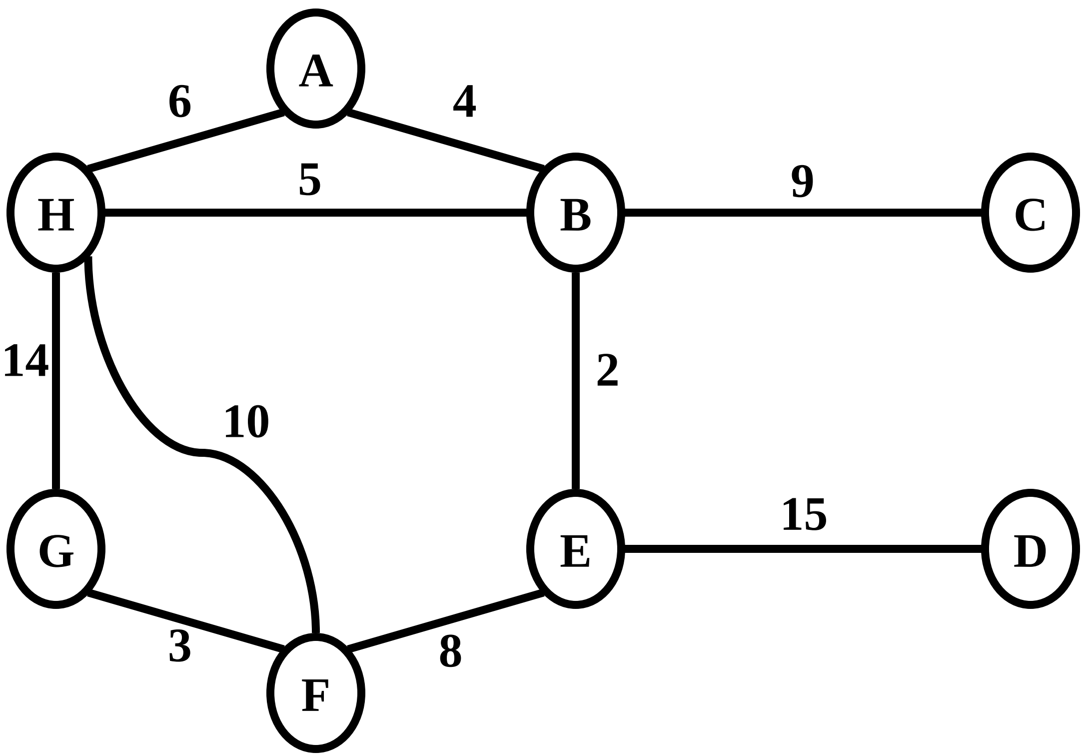
Minimum Spanning Tree

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



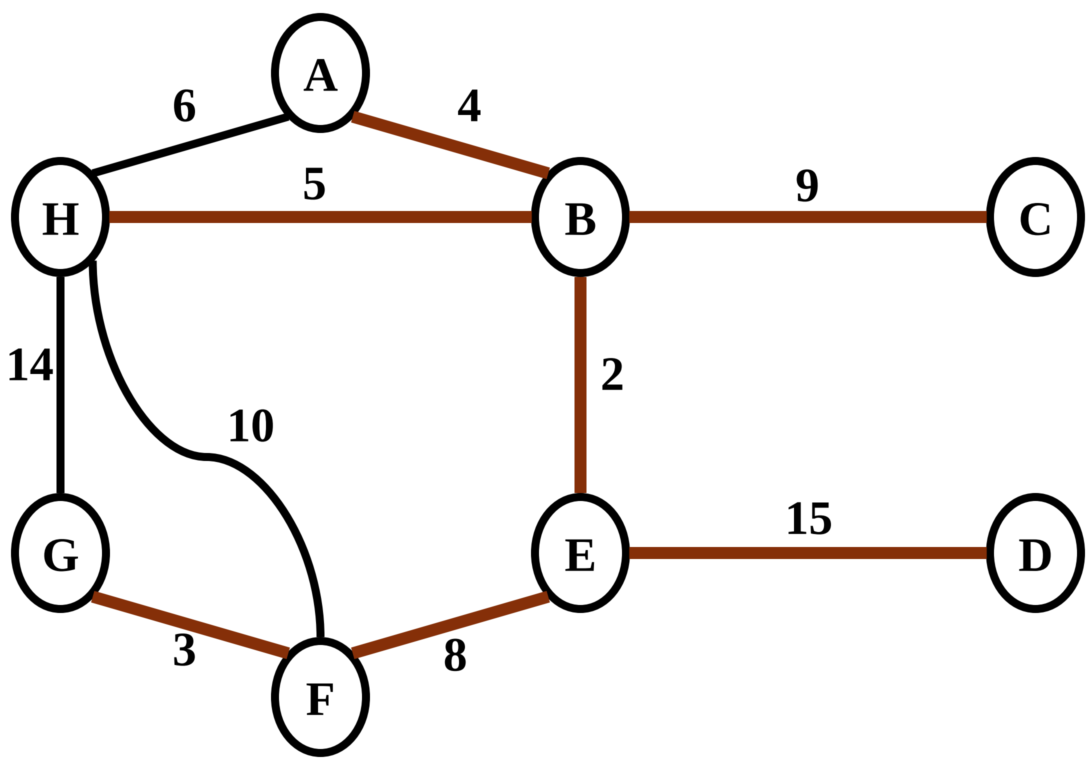
Minimum Spanning Tree

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



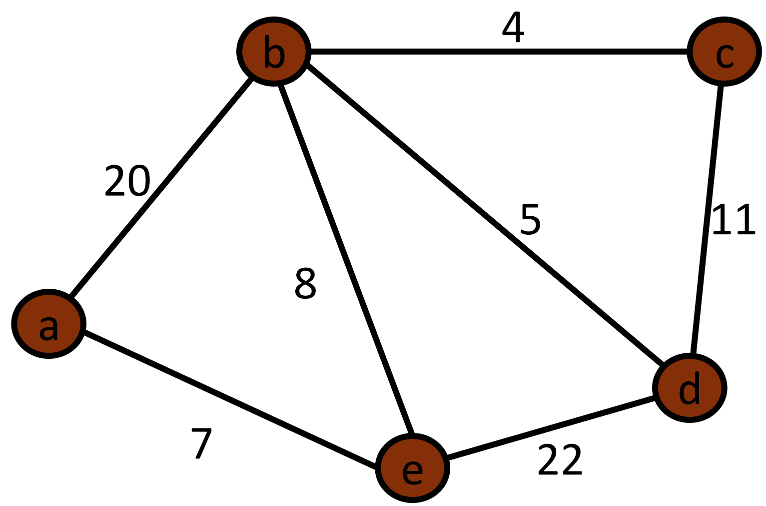
Minimum Spanning Tree

- Answer:



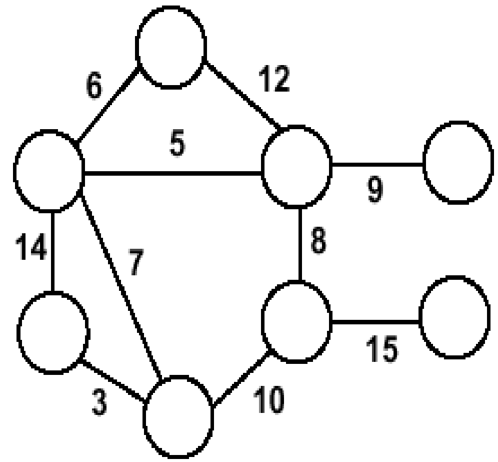
Greedy Algorithms for Minimum Spanning Tree

- [Prim] Extend a tree by including the cheapest out going edge
- [Kruskal] Add the cheapest edge that joins disjoint components



Minimum Spanning Trees

- Undirected, connected graph $G = (V,E)$
- Weight function W: $E \rightarrow R$ (assigning cost or length or other values to edges)

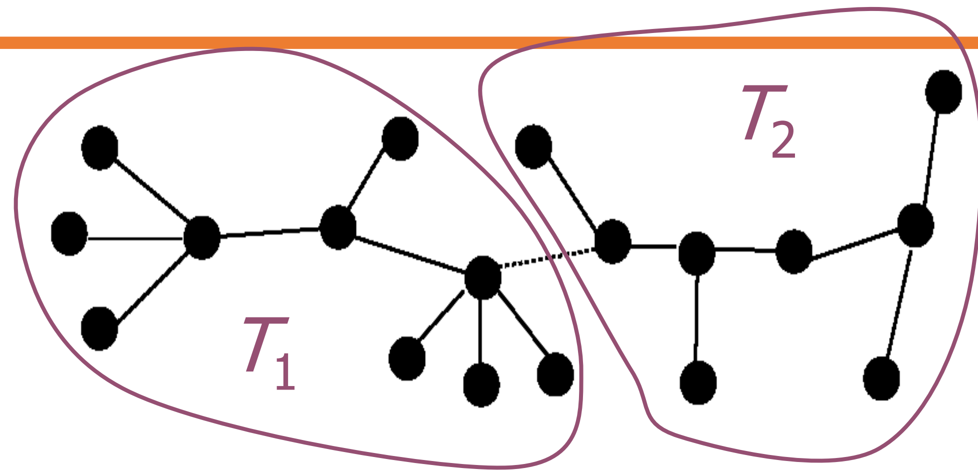


  - Spanning tree: tree that connects all the vertices (above?)
  - Minimum spanning tree: tree that connects all the vertices and minimizes

$$w(T) = \sum_{(u,v)\in T} w(u,v)$$

Optimal Substructure

- MST T



- Removing the edge (u,v) partitions T into T_1 and T_2

$$w(T) = w(u,v) + w(T_1) + w(T_2)$$

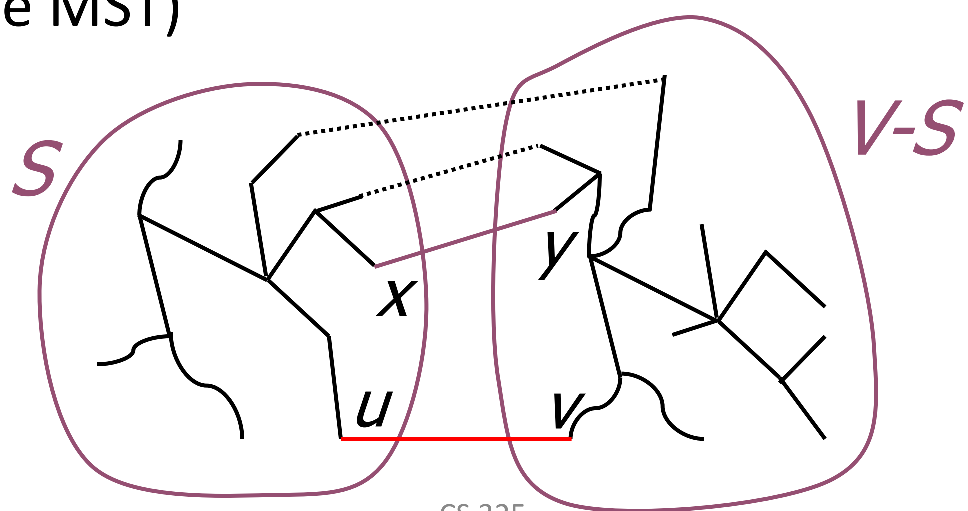
- We claim that T_1 is the MST of $G_1=(V_1,E_1)$, the subgraph of G induced by vertices in T_1
- Also, T_2 is the MST of G_2

Greedy Choice

- Greedy choice property: locally optimal (greedy) choice yields a globally optimal solution
- Theorem
  - Let $G=(V, E)$, and let $S \subseteq V$ and
  - let (u,v) be min-weight edge in G connecting S to $V - S$
  - Then $(u,v) \in T$ – some MST of G

Greedy Choice (2)

- Proof
  - suppose $(u,v) \notin T$
  - look at path from u to v in T
  - swap (x, y) – the first edge on path from u to v in T that crosses from S to $V – S$
  - this improves T – contradiction (T supposed to be MST)



Prim’s Algorithm

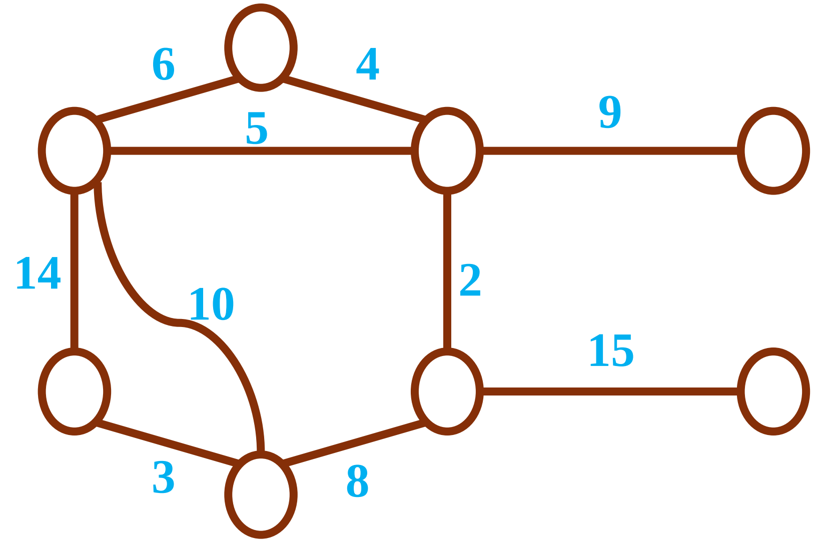
- Vertex based algorithm
- Grows one tree T, **one vertex at a time**
- A cloud covering the portion of T already computed
- Label the vertices *v* outside the cloud with *key*[*v*] – the minimum weigth of an edge connecting *v* to a vertex in the cloud, *key*[*v*] = ∞, if no such edge exists

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

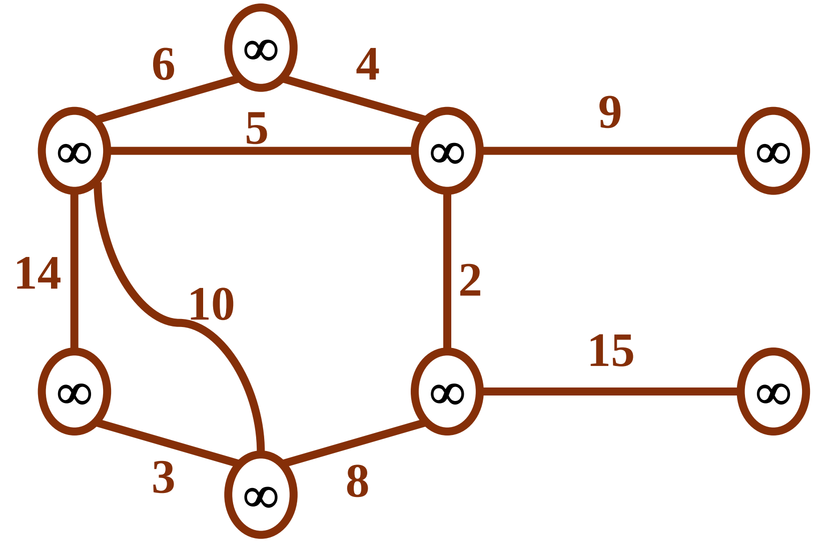
```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



Run on example graph

Prim’s Algorithm

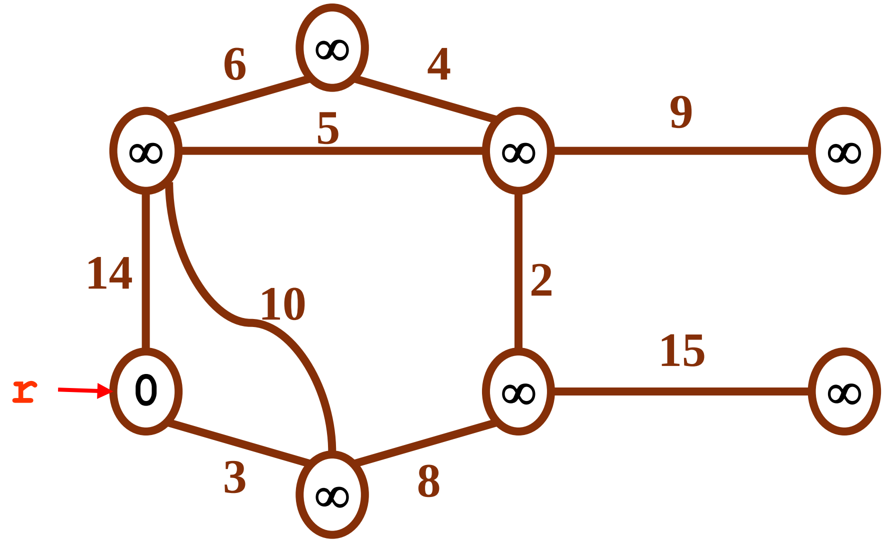
```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



Run on example graph

Prim's Algorithm

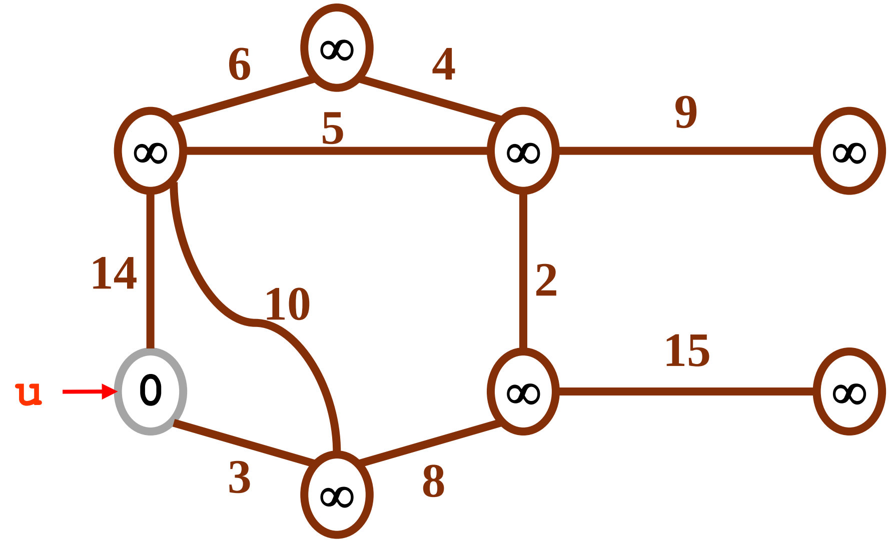
```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



Pick a start vertex r

Prim’s Algorithm

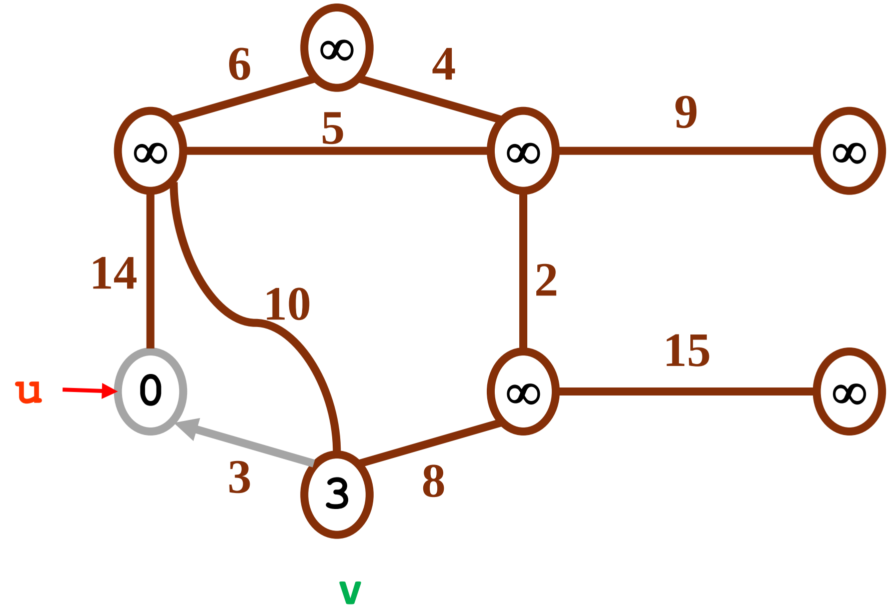
```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



Grey vertices have been removed from Q

Prim's Algorithm

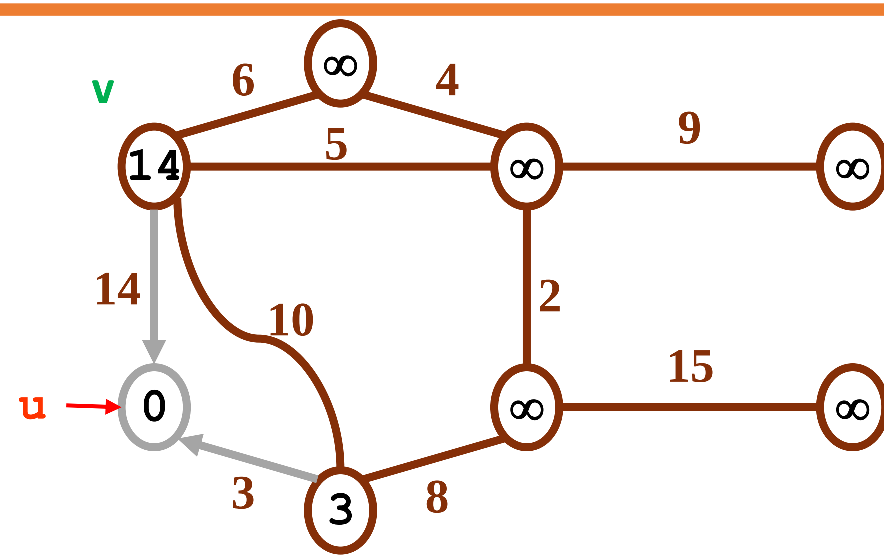
```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



Grey arrows indicate parent pointers

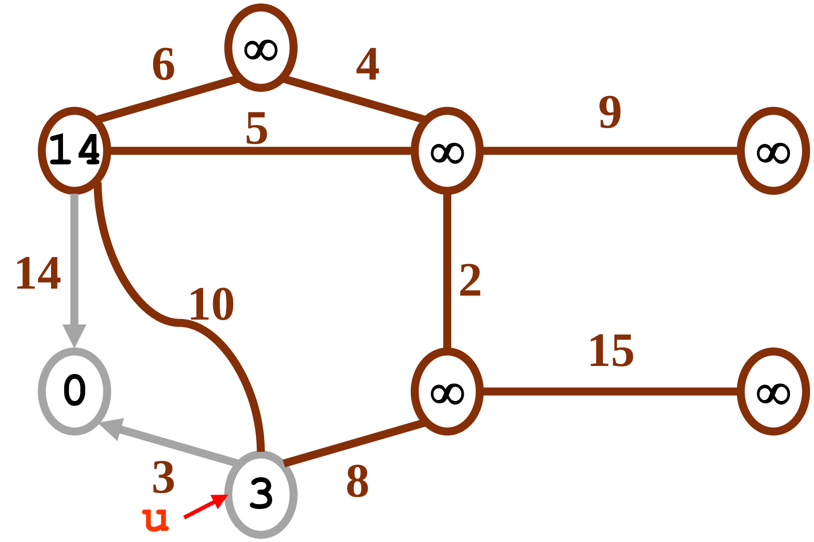
Prim's Algorithm

```
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);
```



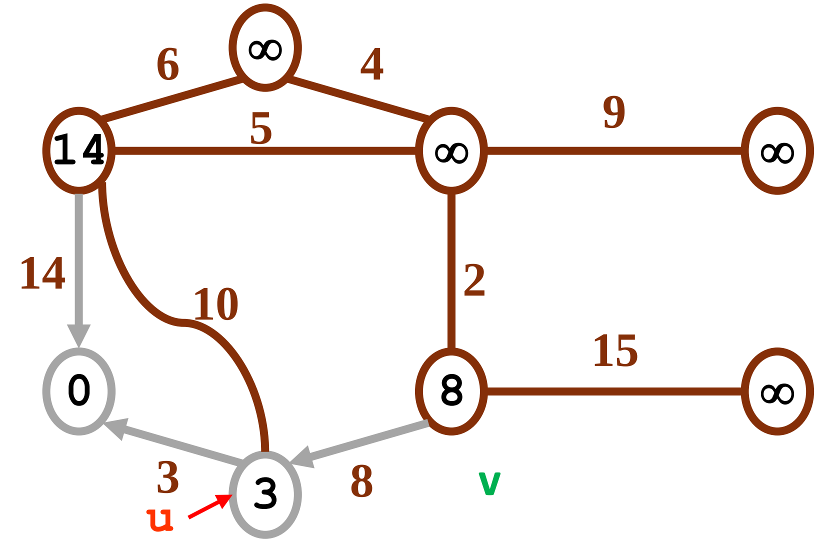
Prim’s Algorithm

```
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);
```



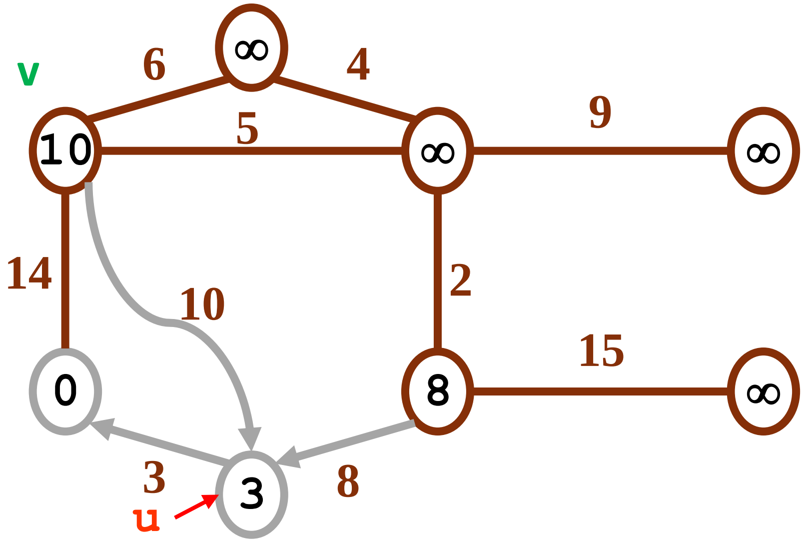
Prim's Algorithm

```
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);
```



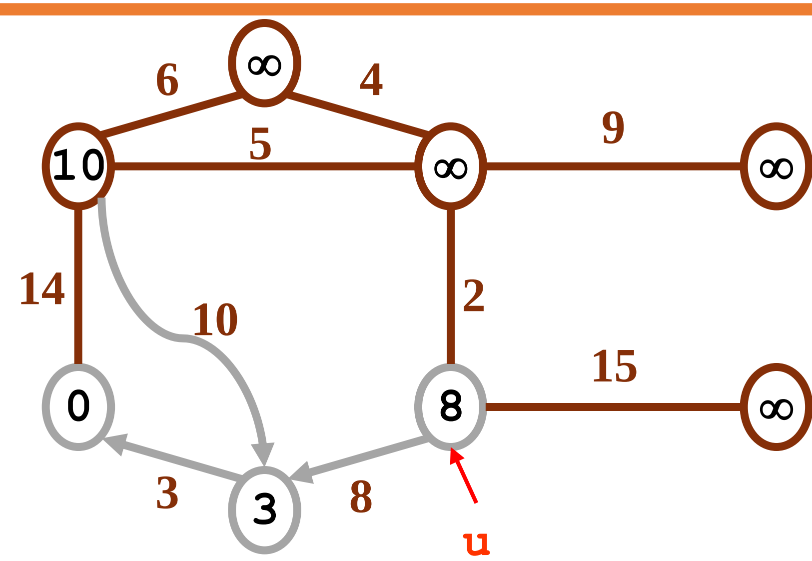
Prim’s Algorithm

```
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);
```



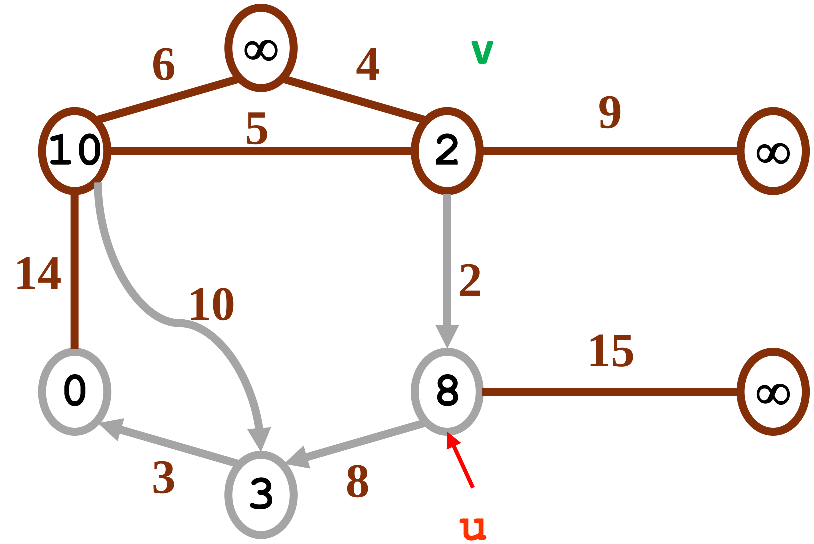
Prim's Algorithm

```
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);
```



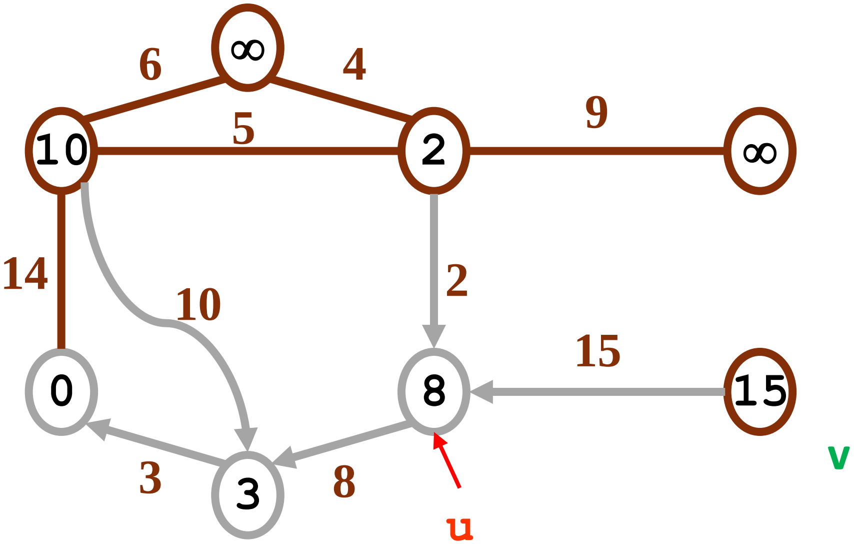
Prim's Algorithm

```
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);
```



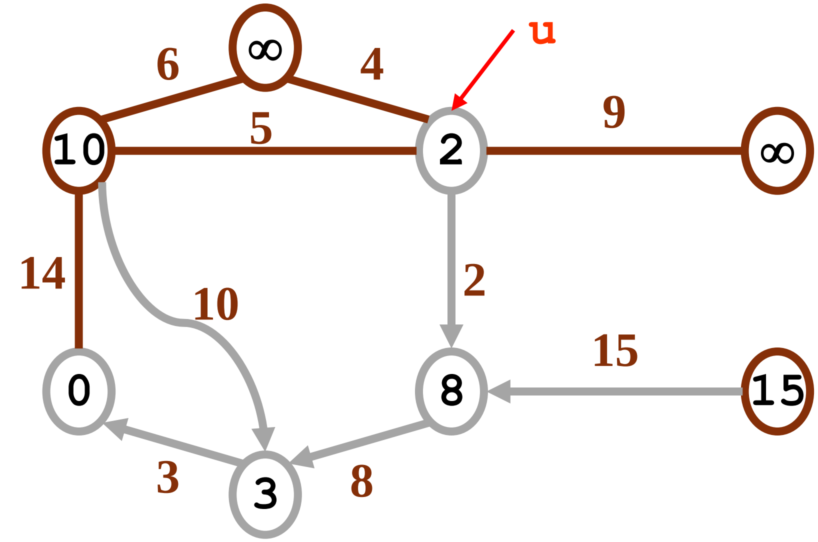
Prim's Algorithm

```
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);
```



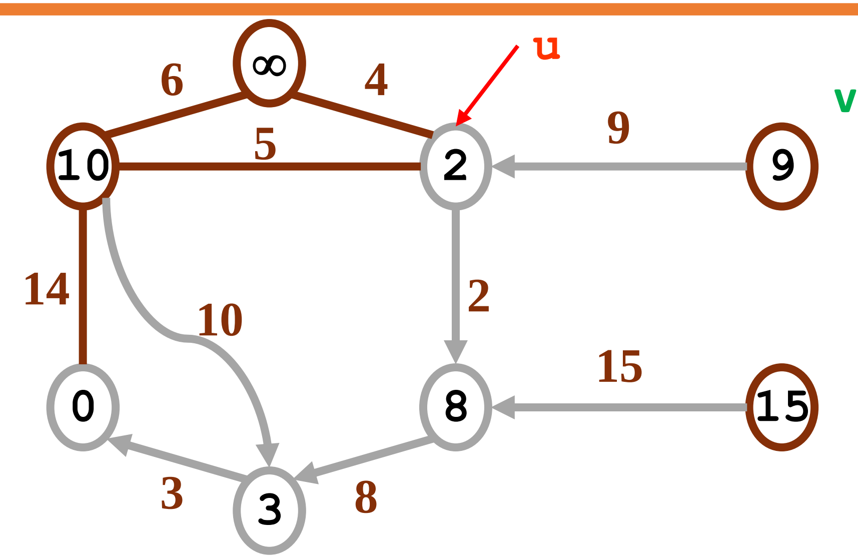
Prim's Algorithm

```
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);
```



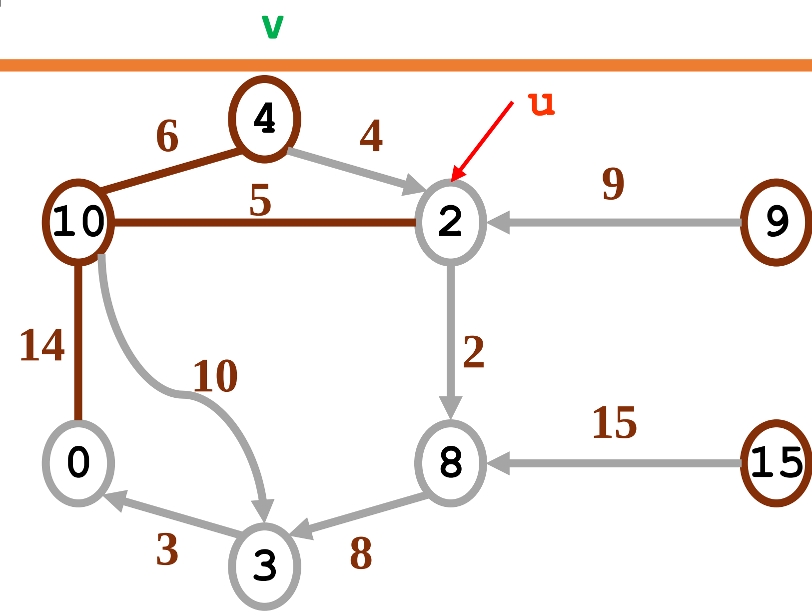
Prim’s Algorithm

```
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);
```



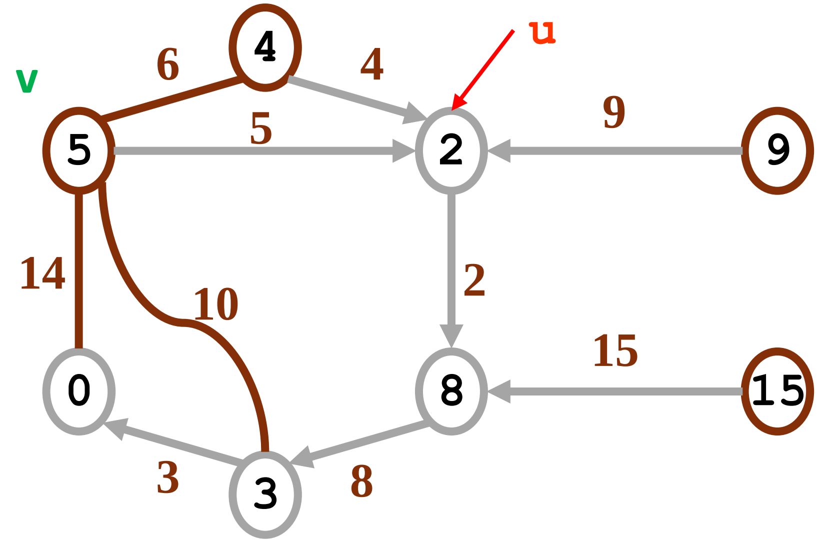
Prim’s Algorithm

```
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);
```



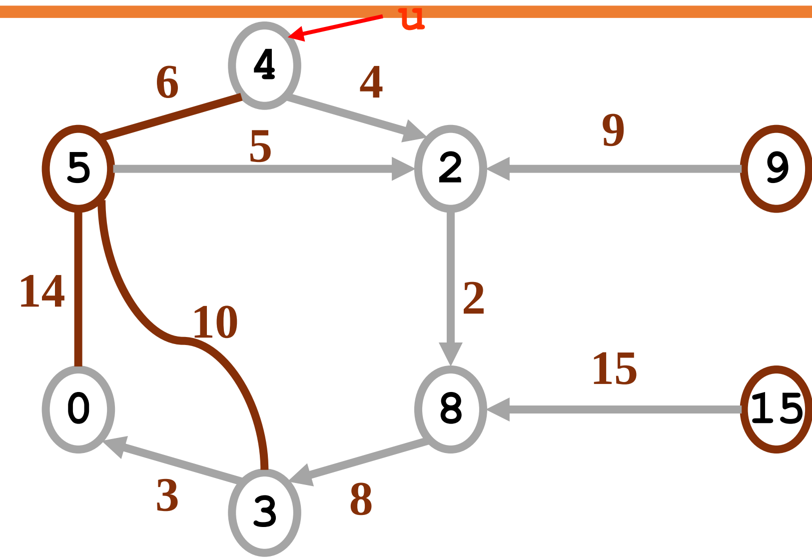
Prim's Algorithm

```
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);
```



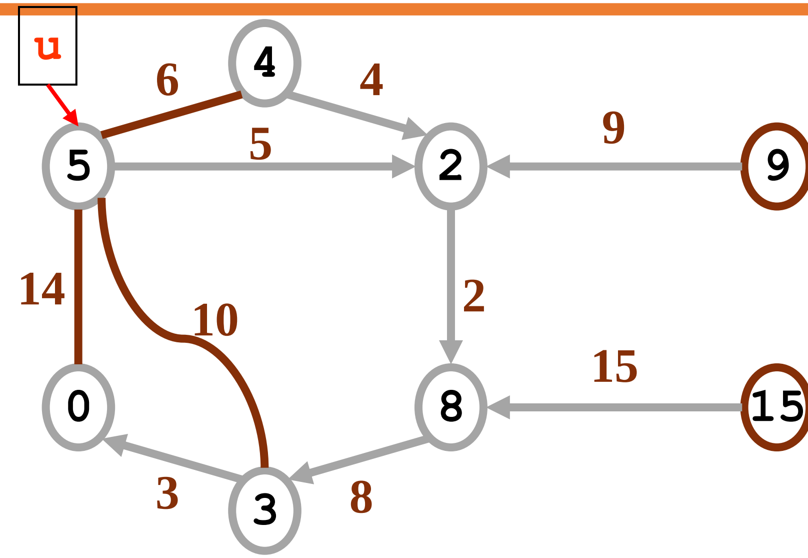
Prim's Algorithm

```
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);
```



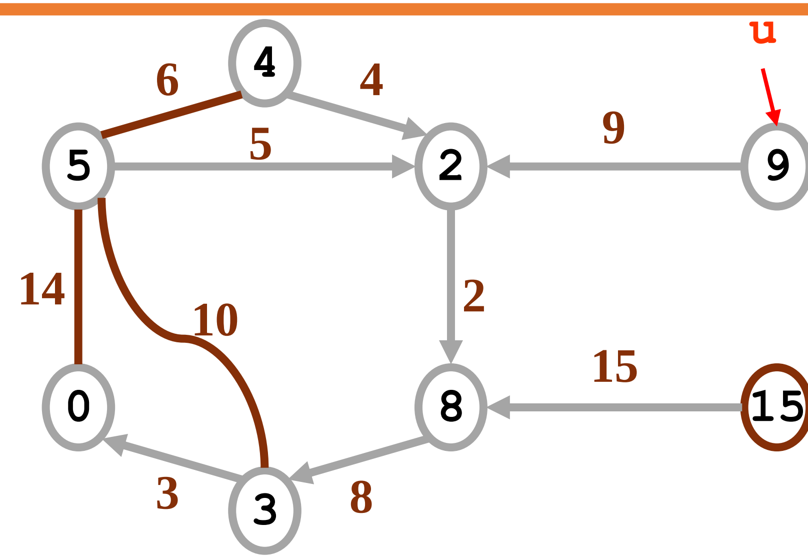
Prim's Algorithm

```
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);
```



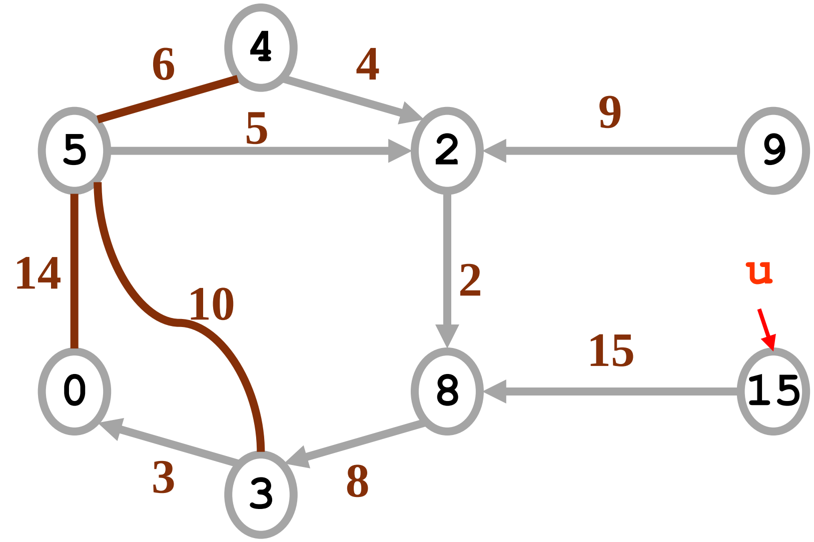
Prim's Algorithm

```
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);
```



Analysis of 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);
```

Analysis of 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));
```

Analysis of 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?

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

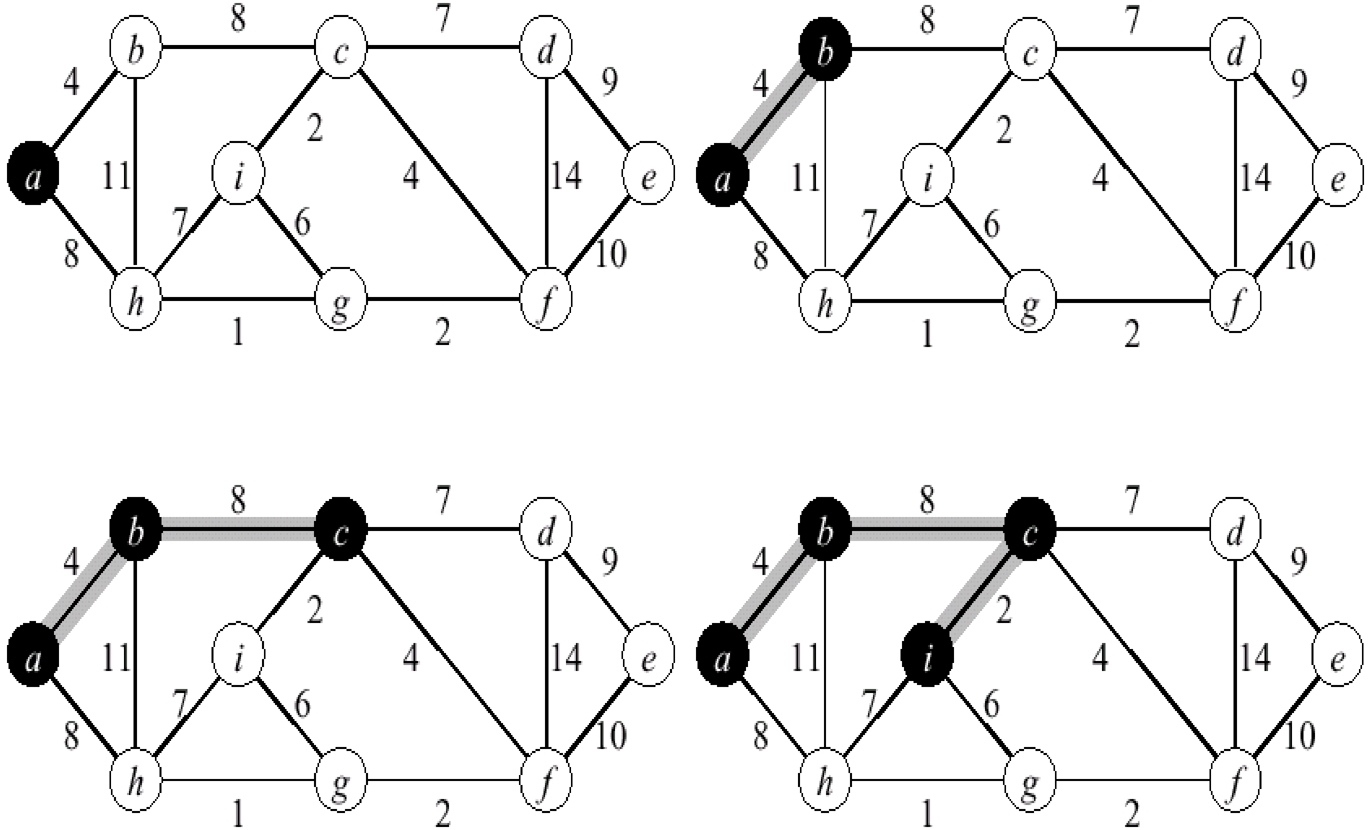
- **binary heap: O(E lg V)**
- **Fibonacci heap: O(V lg V + E)**

Prim's Running Time

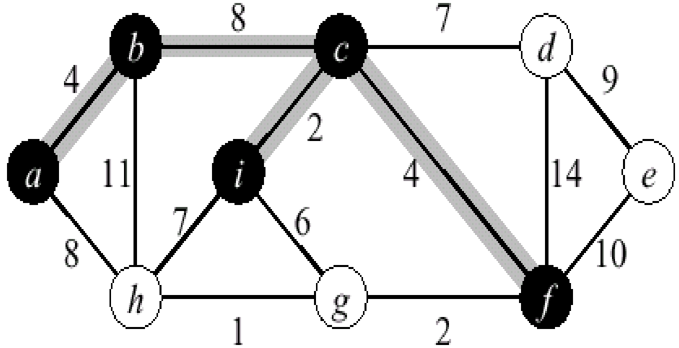
- Time = $|V|\,T(\text{ExtractMin}) + O(E)\,T(\text{ModifyKey})$
- Time = $O(V \lg V + E \lg V) = O(E \lg V)$

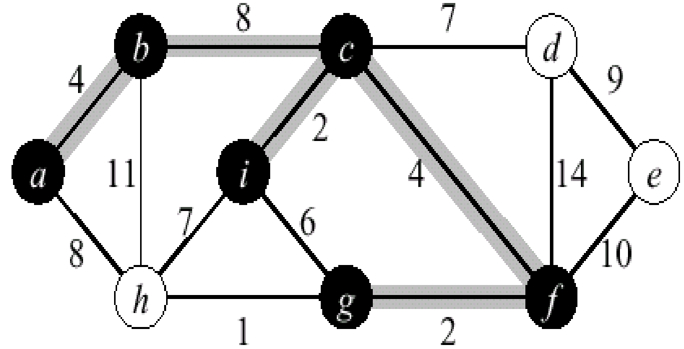
Q	T(ExtractMin)	T(DecreaseKey)	Total
array	$O(V)$	$O(1)$	$O(V^2)$
binary heap	$O(\lg V)$	$O(\lg V)$	$O(E \lg V)$
Fibonacci heap	$O(\lg V)$	$O(1)$ amortized	$O(V \lg V + E)$

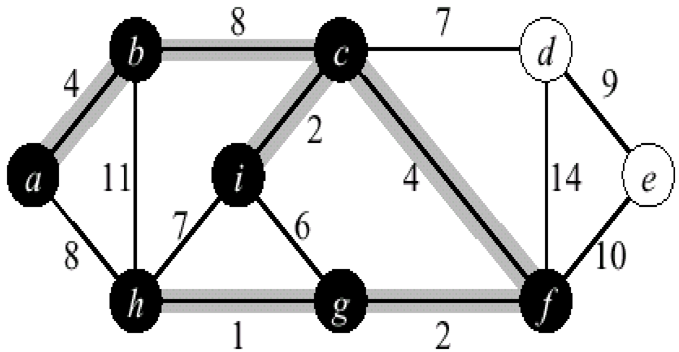
Prim’s Example (2)

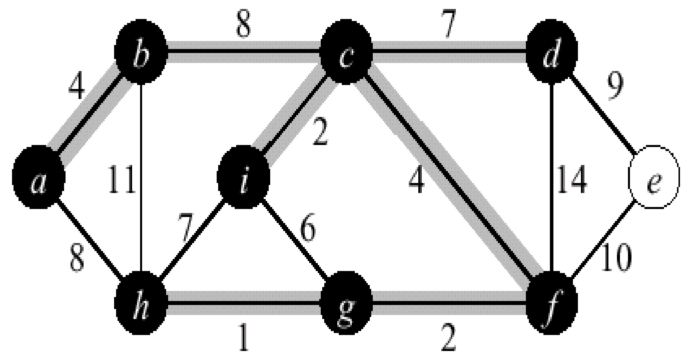


Prim’s Example (2)

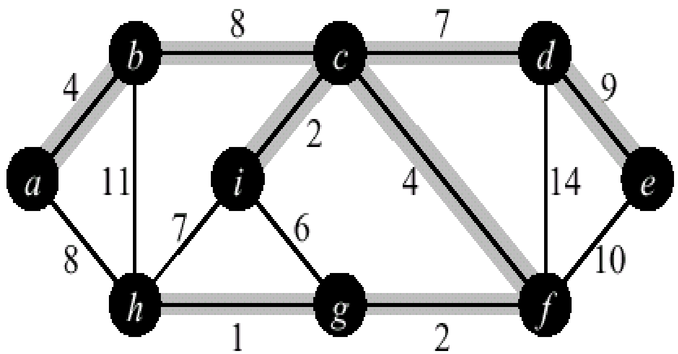








Prim’s Example (2)



Kruskal's Algorithm

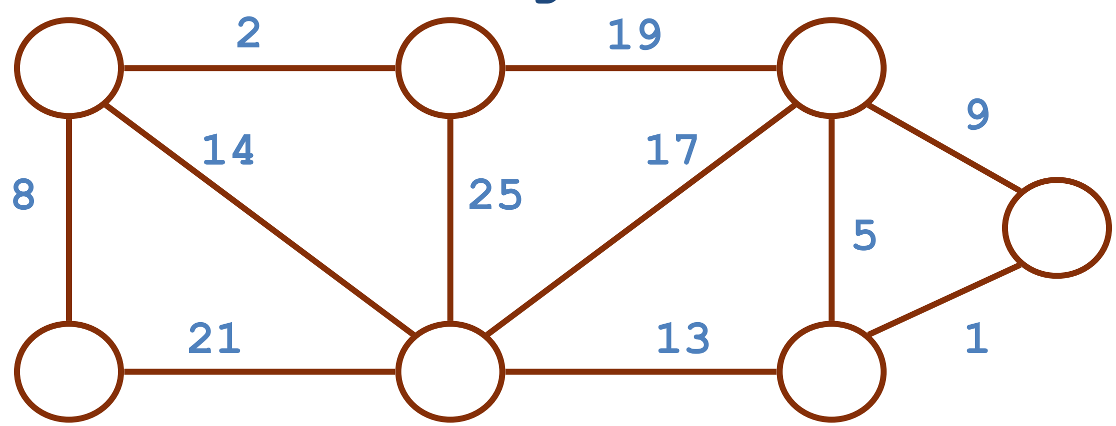
- Edge based algorithm
- Add the edges one at a time, in increasing weight order
- The algorithm maintains *A* – a **forest of trees**. An edge is accepted it if connects vertices of distinct trees
- We need an Abstract Data Type (ADT) that maintains a partition, i.e.,a collection of disjoint sets
  - MakeSet(S,x): $S \leftarrow S \cup \{\{x\}\}$
  - Union(S_i,S_j): $S \leftarrow S - \{S_i, S_j\} \cup \{S_i \cup S_j\}$
  - FindSet(S, x): returns unique $S_i \in S$, where $x \in S_i$

Kruskal's Algorithm

```
Kruskal()
{
   A = ∅;
   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)
         A = A U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

Kruskal’s Algorithm

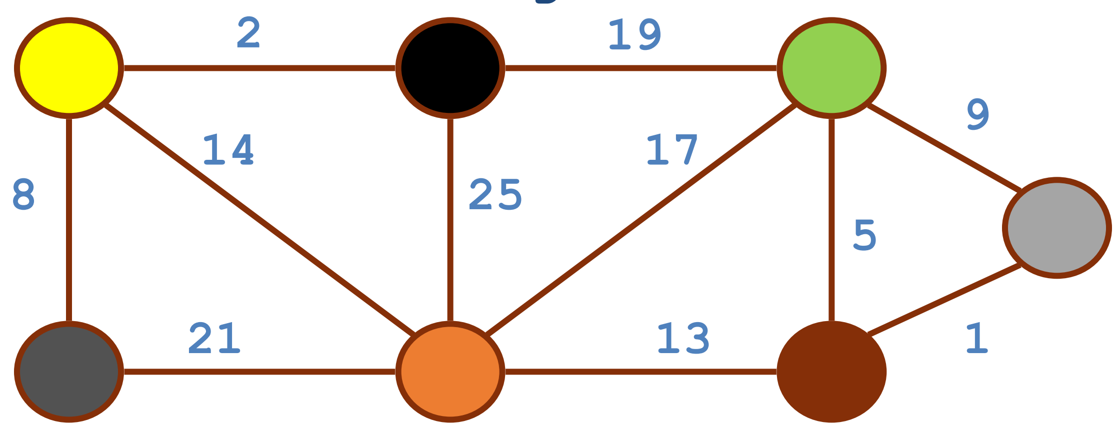
Run the algorithm:




```
Kruskal()
{
    A = ∅;
    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)
            A = A U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

Kruskal's Algorithm

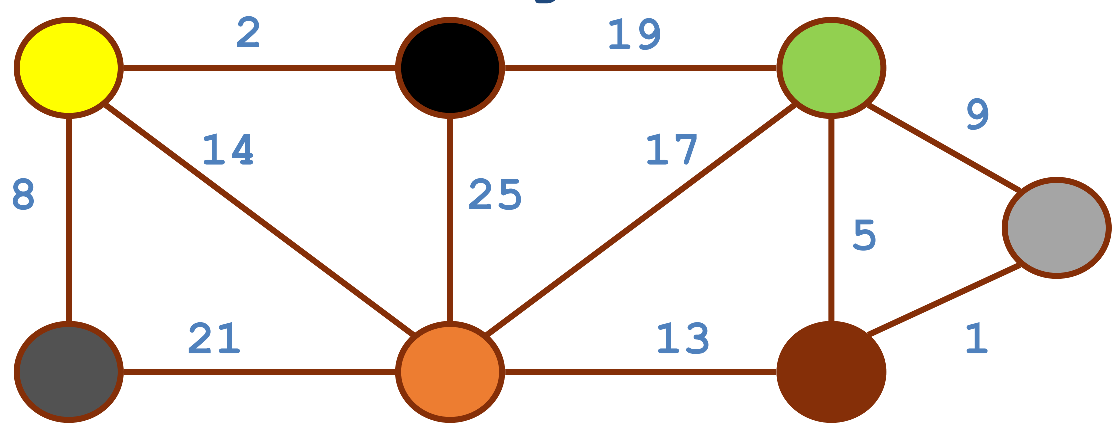
Run the algorithm:




```
Kruskal()
{
    A = ∅;
    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)
            A = A U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

Kruskal's Algorithm

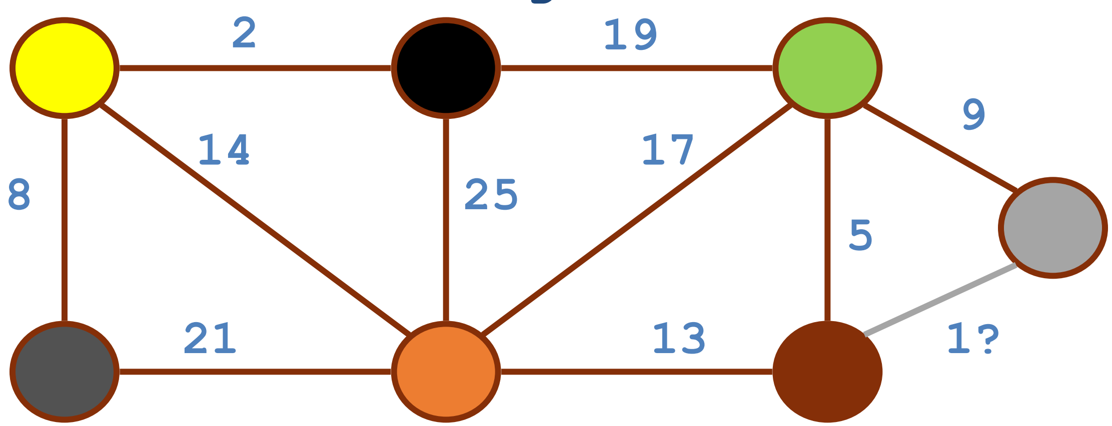
Run the algorithm:




```
Kruskal()
{
    A = ∅;
    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)
            A = A U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

Kruskal's Algorithm

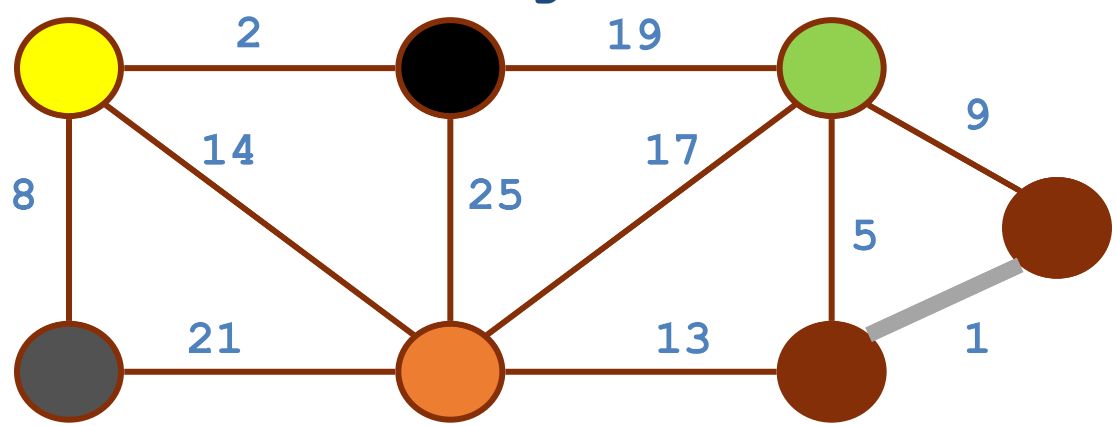
Run the algorithm:




```
Kruskal()
{
   A = ∅;
   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)
         A = A U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

Kruskal's Algorithm

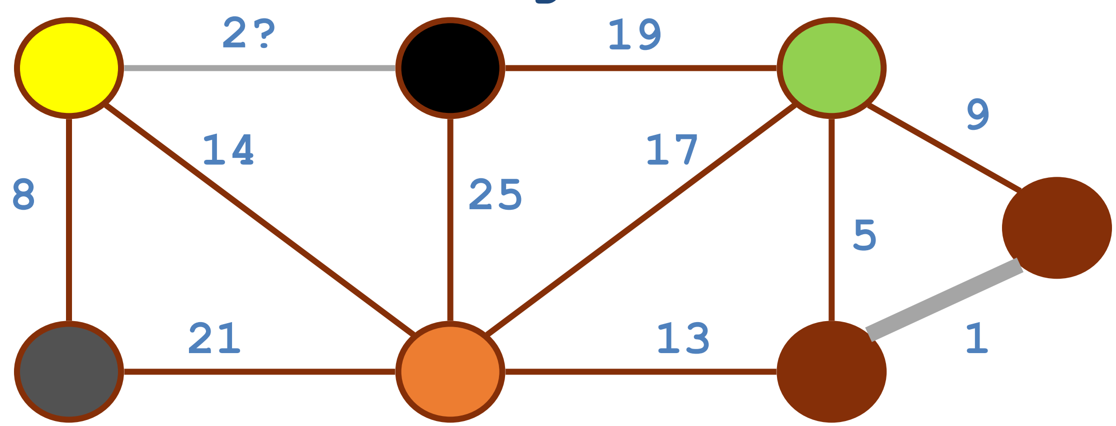
Run the algorithm:




```
Kruskal()
{
    A = ∅;
    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)
            A = A U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

Kruskal’s Algorithm

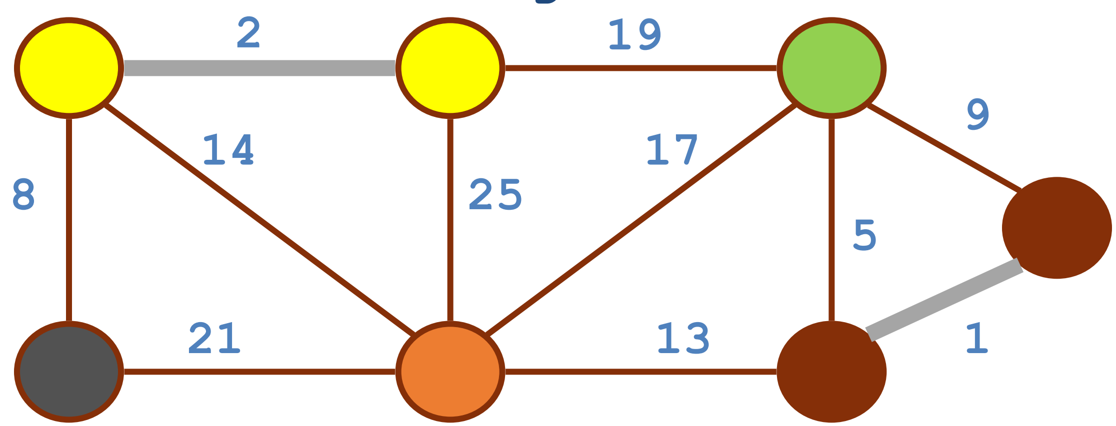
Run the algorithm:




```
Kruskal()
{
    A = ∅;
    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)
            A = A U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

Kruskal's Algorithm

Run the algorithm:

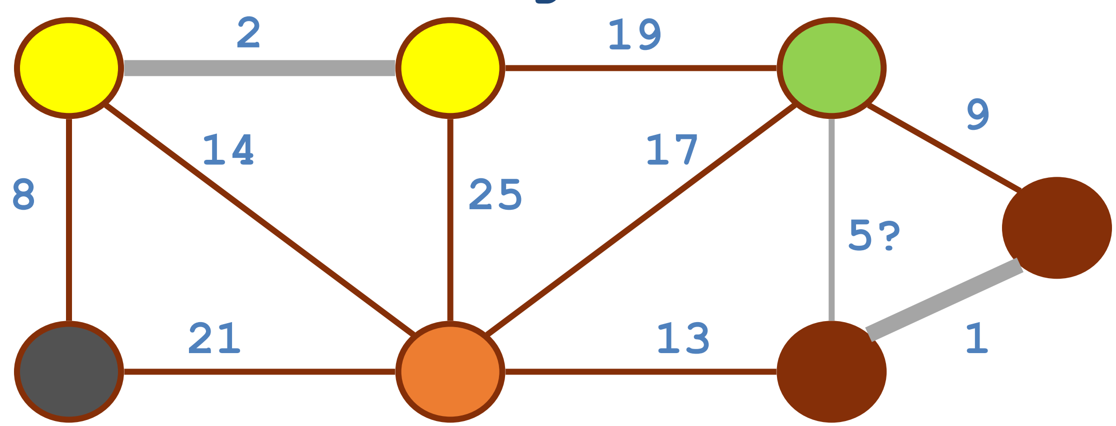



```
Kruskal()
{
    A = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            A = A U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

Kruskal's Algorithm

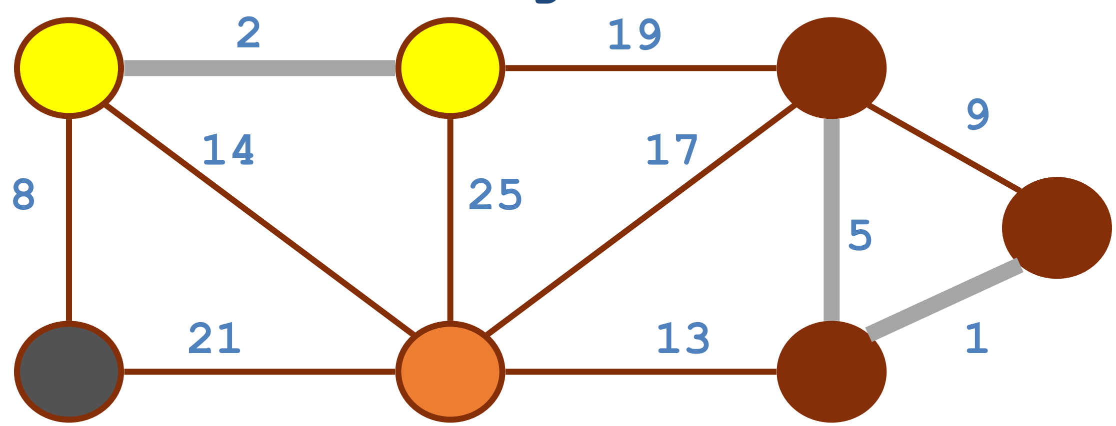
```
Kruskal()
{
    A = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            A = A U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

Run the algorithm:



Kruskal's Algorithm

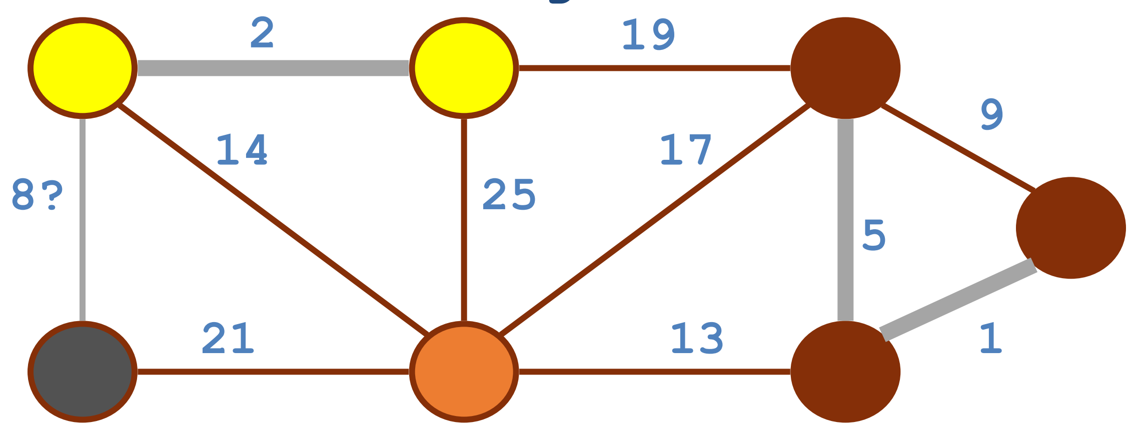
Run the algorithm:




```
Kruskal()
{
   A = ∅;
   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)
         A = A U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

Kruskal's Algorithm

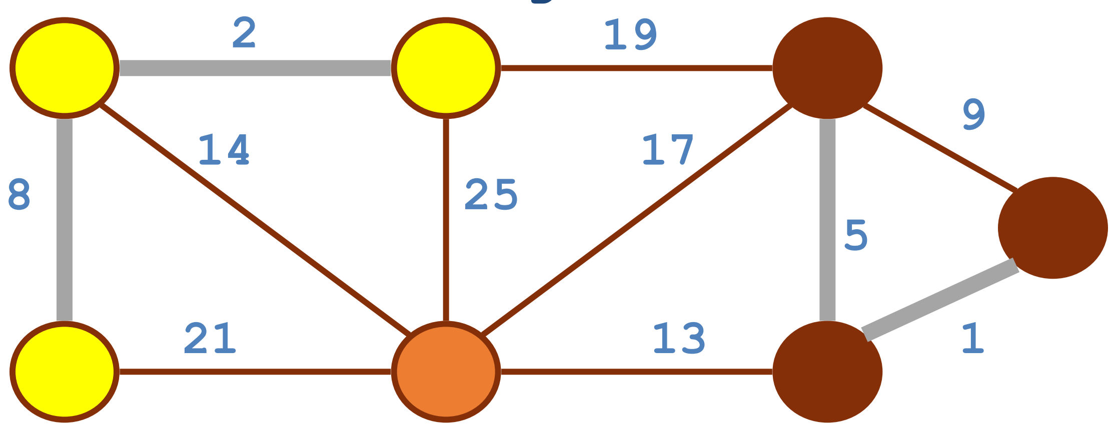
Run the algorithm:




```
Kruskal()
{
    A = ∅;
    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)
            A = A U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

Kruskal's Algorithm

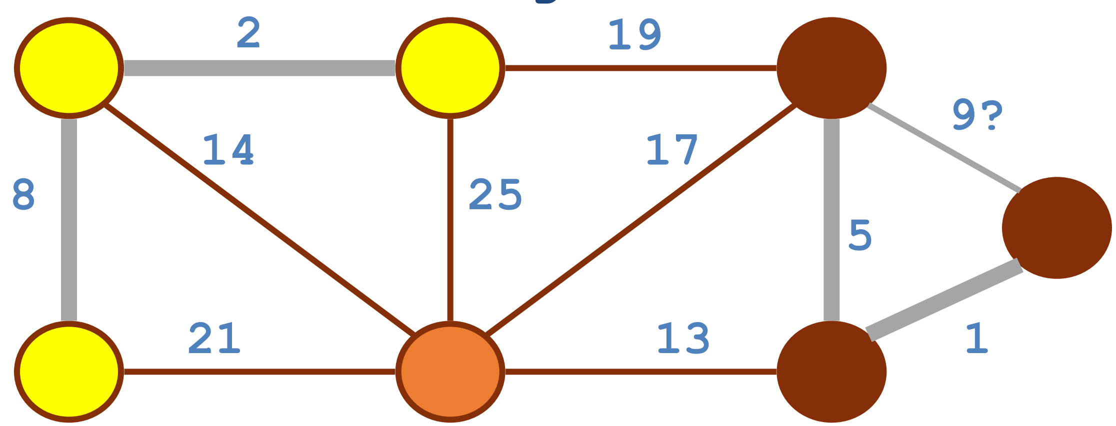
Run the algorithm:




```
Kruskal()
{
    A = ∅;
    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)
            A = A U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

Kruskal's Algorithm

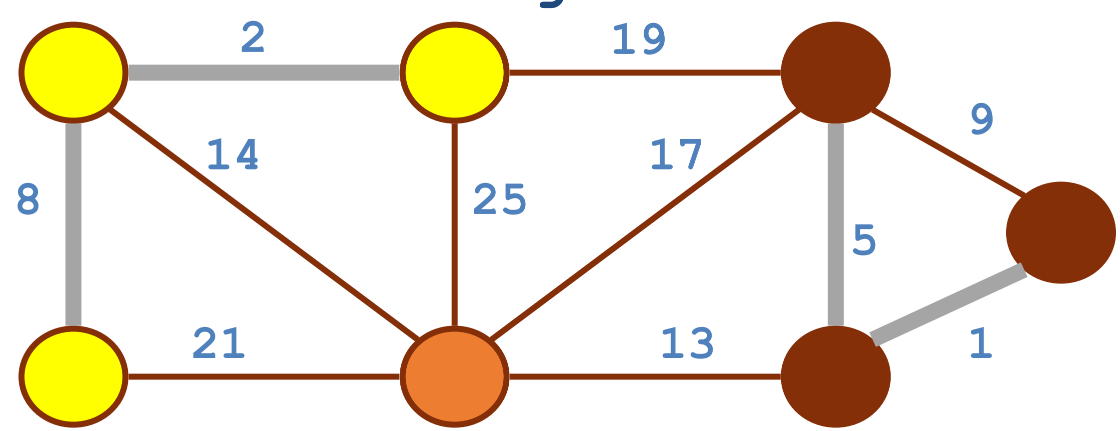
Run the algorithm:




```
Kruskal()
{
   A = ∅;
   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)
         A = A U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

Kruskal's Algorithm

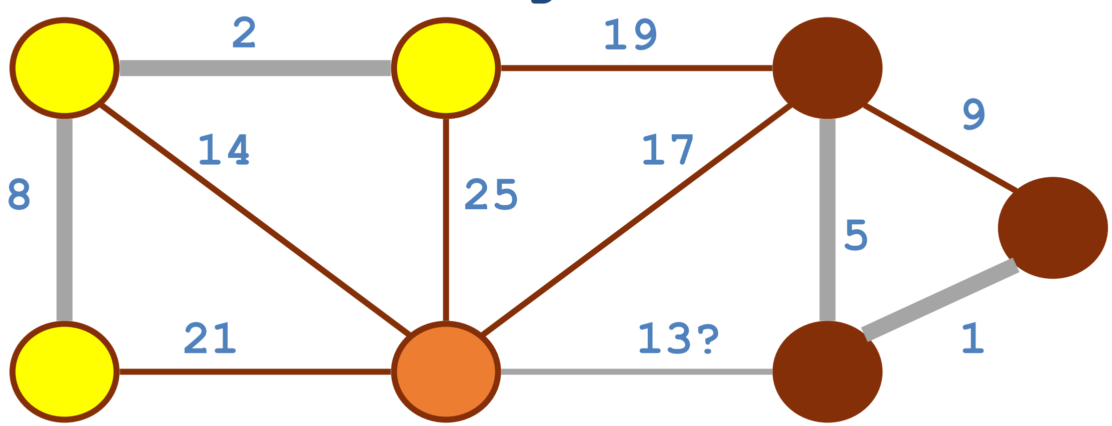
Run the algorithm:




```
Kruskal()
{
    A = ∅;
    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)
            A = A U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

Kruskal’s Algorithm

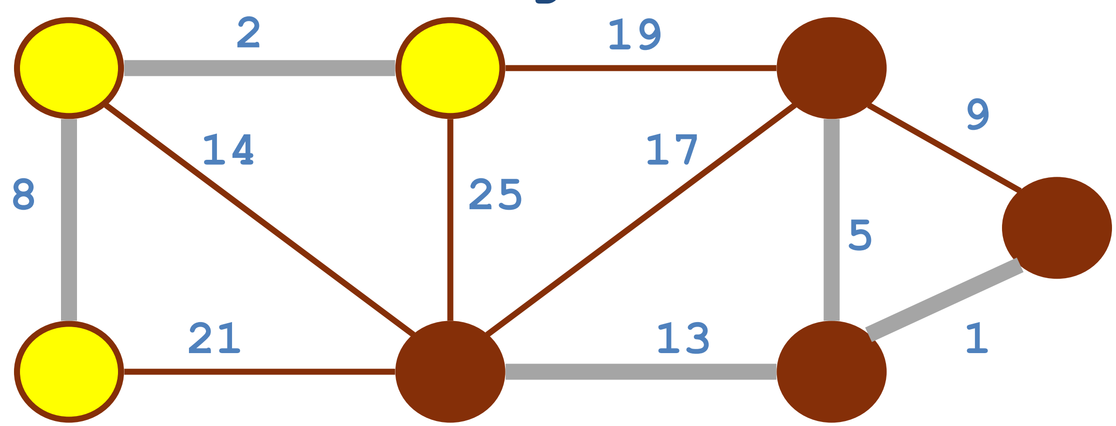
Run the algorithm:




```
Kruskal()
{
    A = ∅;
    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)
            A = A U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

Kruskal's Algorithm

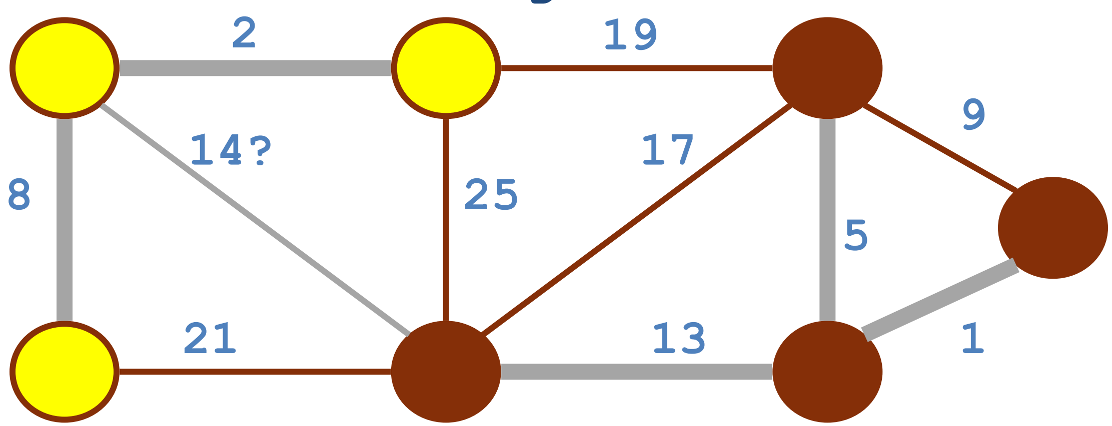
Run the algorithm:




```
Kruskal()
{
    A = ∅;
    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)
            A = A U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

Kruskal's Algorithm

Run the algorithm:

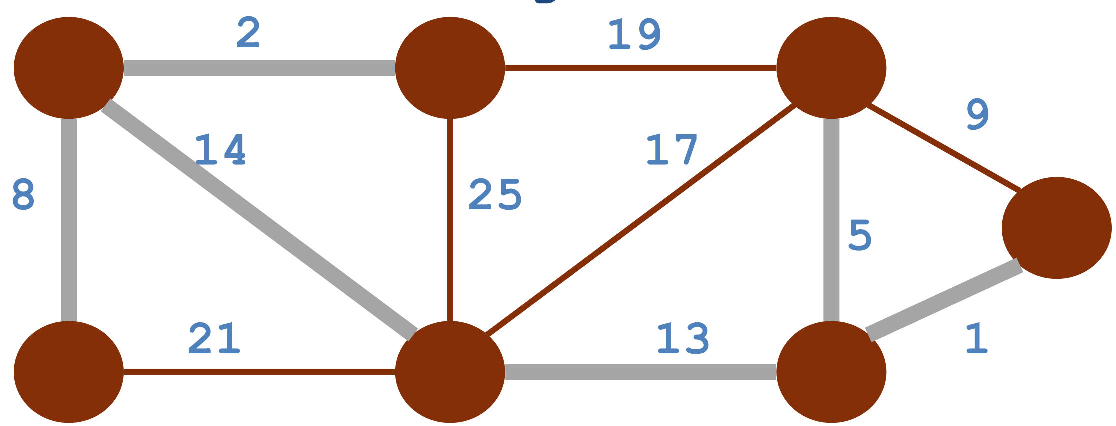



```
Kruskal()
{
    A = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            A = A U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

Kruskal’s Algorithm

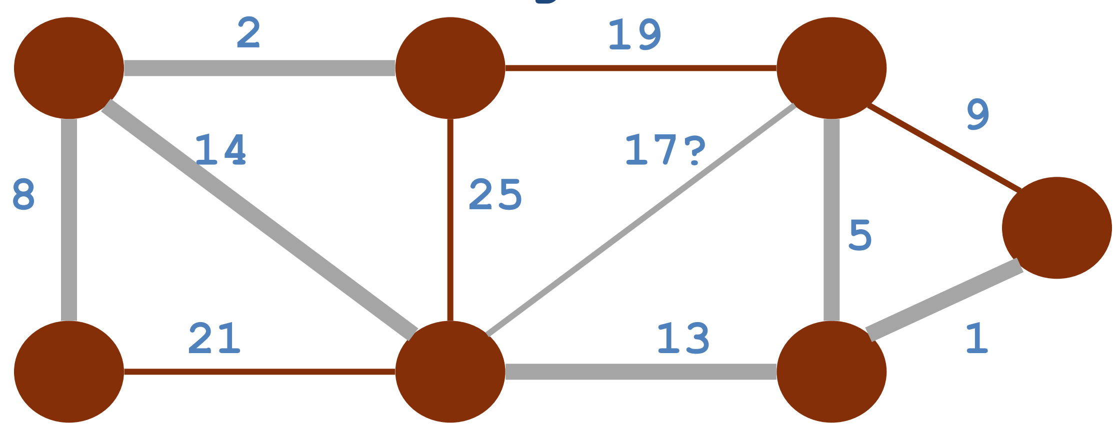
```
Kruskal()
{
    A = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            A = A U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

Run the algorithm:



Kruskal's Algorithm

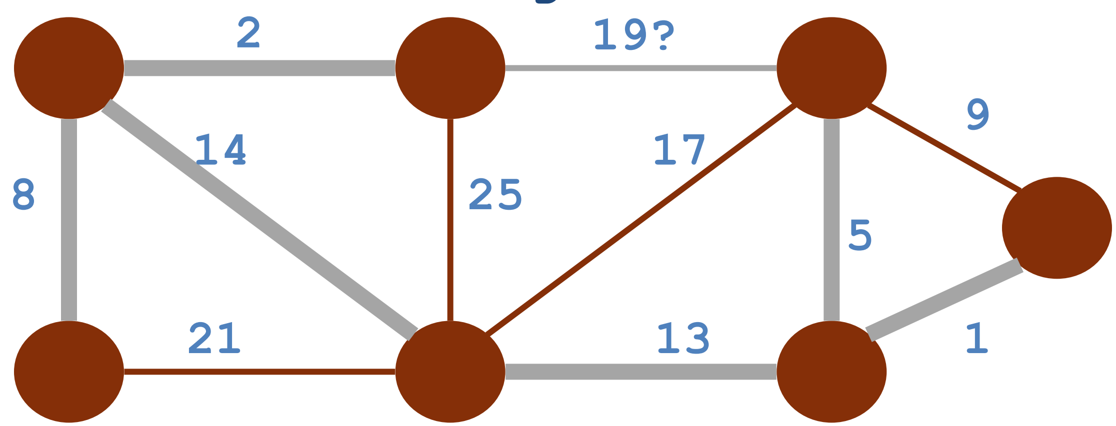
Run the algorithm:




```
Kruskal()
{
    A = ∅;
    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)
            A = A U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

Kruskal's Algorithm

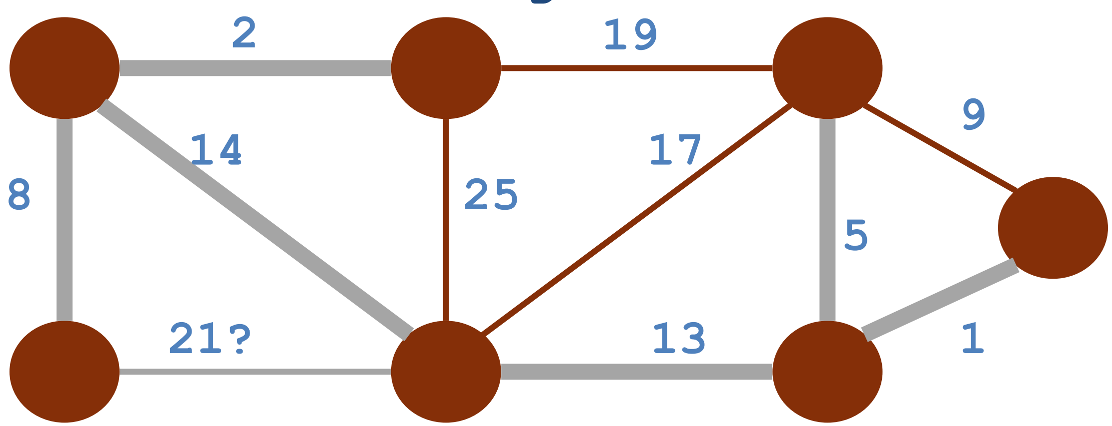
Run the algorithm:




```
Kruskal()
{
   A = ∅;
   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)
         A = A U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

Kruskal's Algorithm

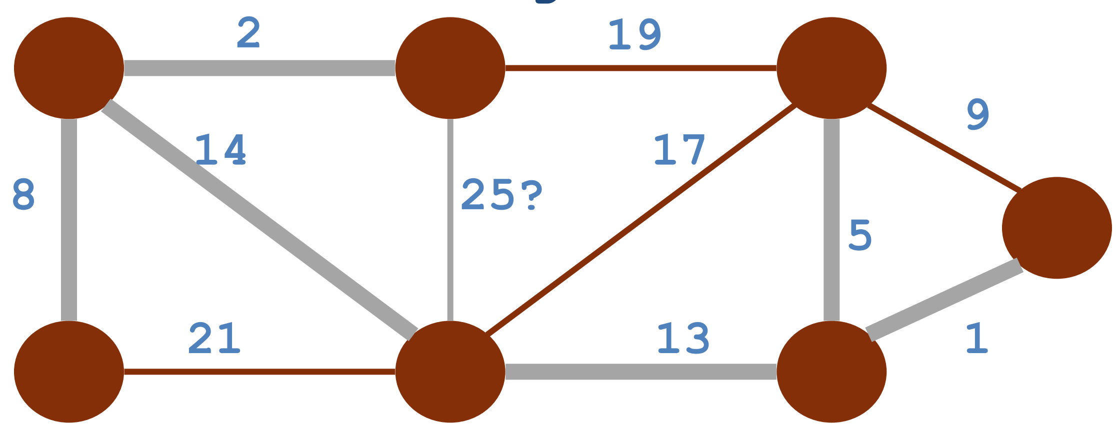
Run the algorithm:




```
Kruskal()
{
    A = ∅;
    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)
            A = A U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

Kruskal's Algorithm

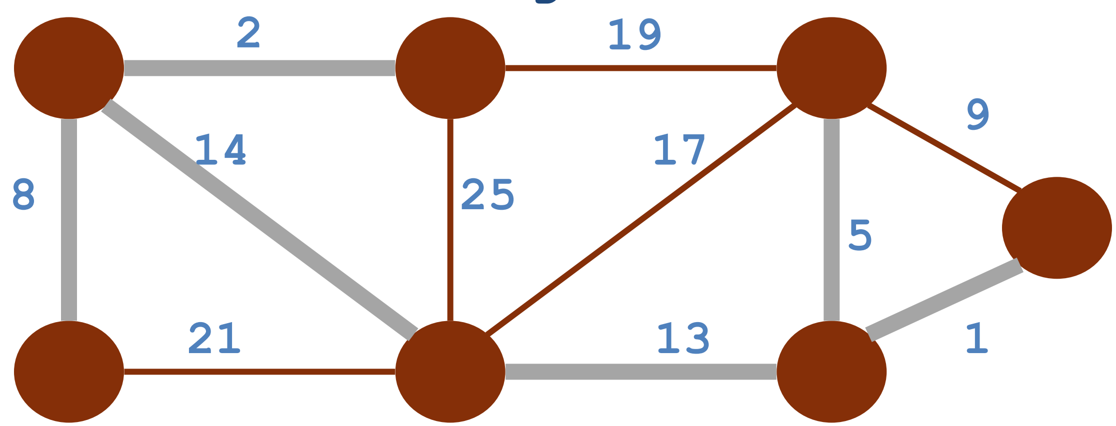
Run the algorithm:




```
Kruskal()
{
   A = ∅;
   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)
         A = A U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

Kruskal's Algorithm

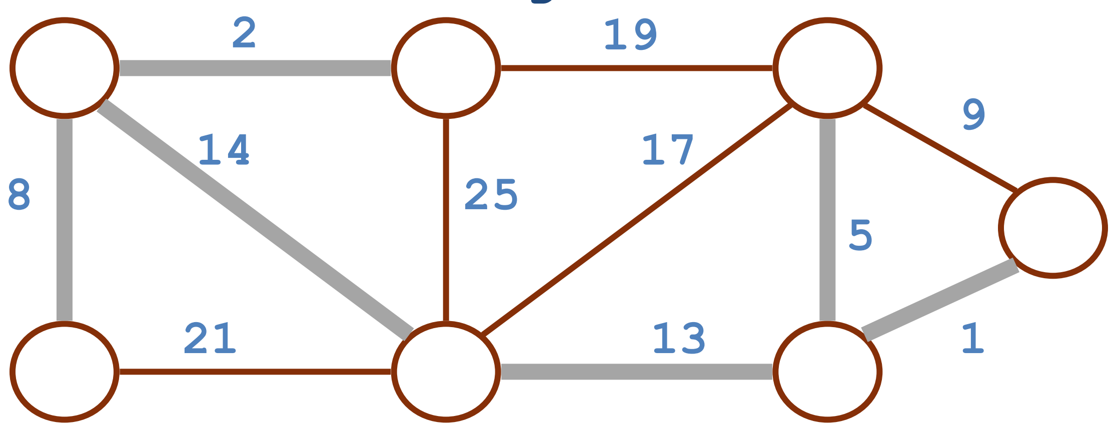
Run the algorithm:




```
Kruskal()
{
    A = ∅;
    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)
            A = A U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

Kruskal's Algorithm

Run the algorithm:




```
Kruskal()
{
    A = ∅;
    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)
            A = A U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

Correctness Of Kruskal's Algorithm

- Sketch of a proof that this algorithm produces an MST for *T*:
  - Assume algorithm is wrong: result is not an MST
  - Then algorithm adds a wrong edge at some point
  - If it adds a wrong edge, there must be a lower weight edge (cut and paste argument)
  - But algorithm chooses lowest weight edge at each step. Contradiction
- Again, important to be comfortable with cut and paste arguments

Kruskal's Algorithm

What will affect the running time?

```
Kruskal()
{
   A = ∅;
   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)
         A = A U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

Kruskal’s Algorithm

What will affect the running time?

1 Sort

O(V) MakeSet() calls

O(E) FindSet() calls

O(E) Union() calls

```
Kruskal()
{
    A = ∅;
    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)
            A = A U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

Kruskal's Algorithm: Running Time

- To summarize:
  - Sort edges: $O(E \lg E)$
  - $O(V)$ MakeSet()'s
  - $O(E)$ FindSet()'s
  - $O(E)$ Union()'s
- Upshot:
  - Best disjoint-set union algorithm makes above 3 operations take $O(E \cdot \alpha(E,V))$, α almost constant
  - Overall thus $O(E \lg E) = O(E \lg V)$ since $E < V^2$

Kruskal Running Time

- Initialization $O(V)$ time
- Sorting the edges $\Theta(E \lg E) = \Theta(E \lg V)$
- $O(E)$ calls to FindSet
- Union costs
  - Let *t(v)* – the number of times *v* is moved to a new cluster
  - Each time a vertex is moved to a new cluster the size of the cluster containing the vertex at least doubles: $t(v) \leq \log V$
  - Total time spent doing Union $$\sum_{v \in V} t(v) \leq |V| \log |V|$$
- Total time: $O(E \lg V)$