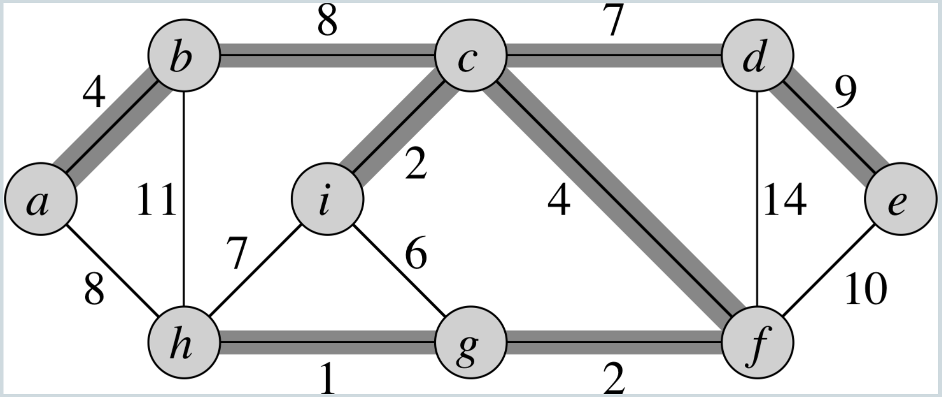
CS146: Data Structures and Algorithms
Lecture 14

MINIMUM SPANNING TREE
PRIM'S AND KRUSKAL'S ALGORITHMS (CH 23)

INSTRUCTOR: KATERINA POTIKA
CS SJSU

Minimum Spanning Tree (Ch 23)

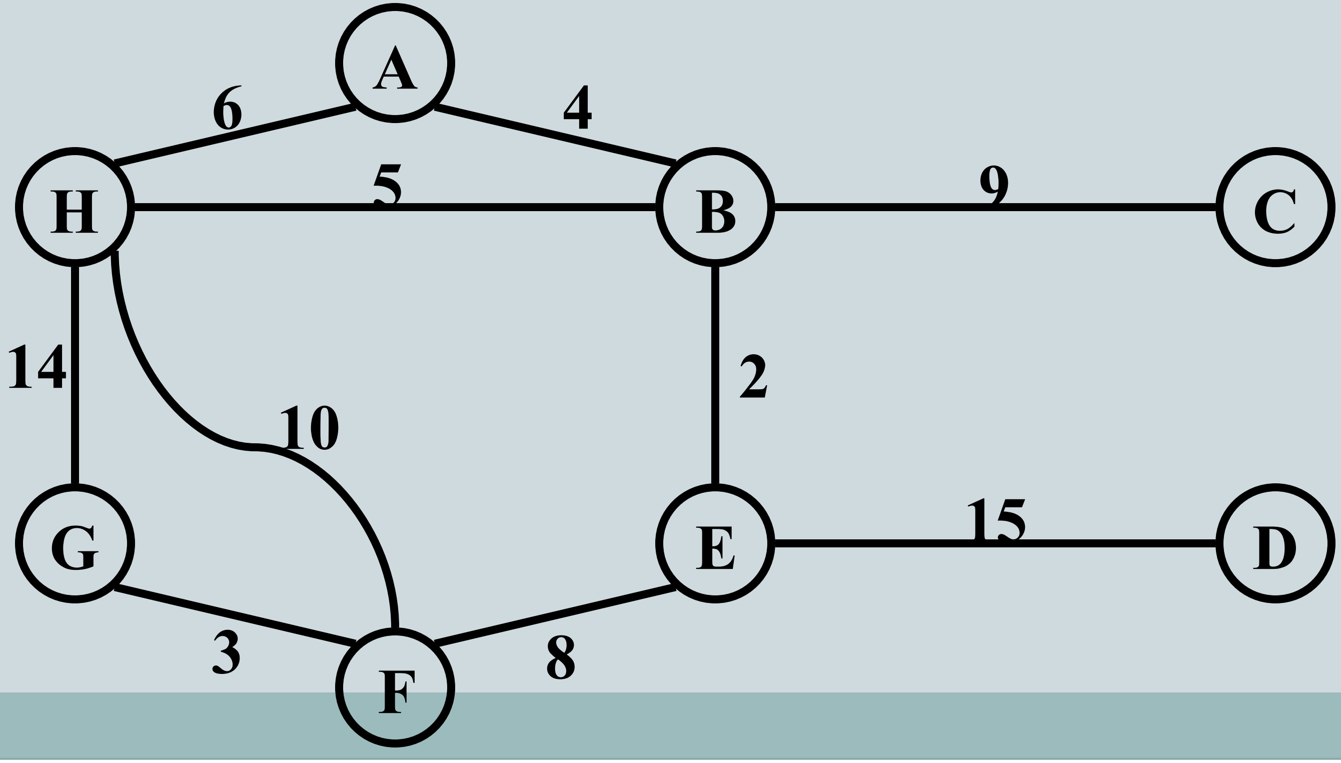
- Given: a connected, undirected, weighted graph with
- w: E-> R
- Output: a *spanning tree T (spans all vertices)*
- Goal: minimize the total weight of the selected edges
  w(T)=∑(u,v)∈T↑▒w(u,v)



- **Applications?**

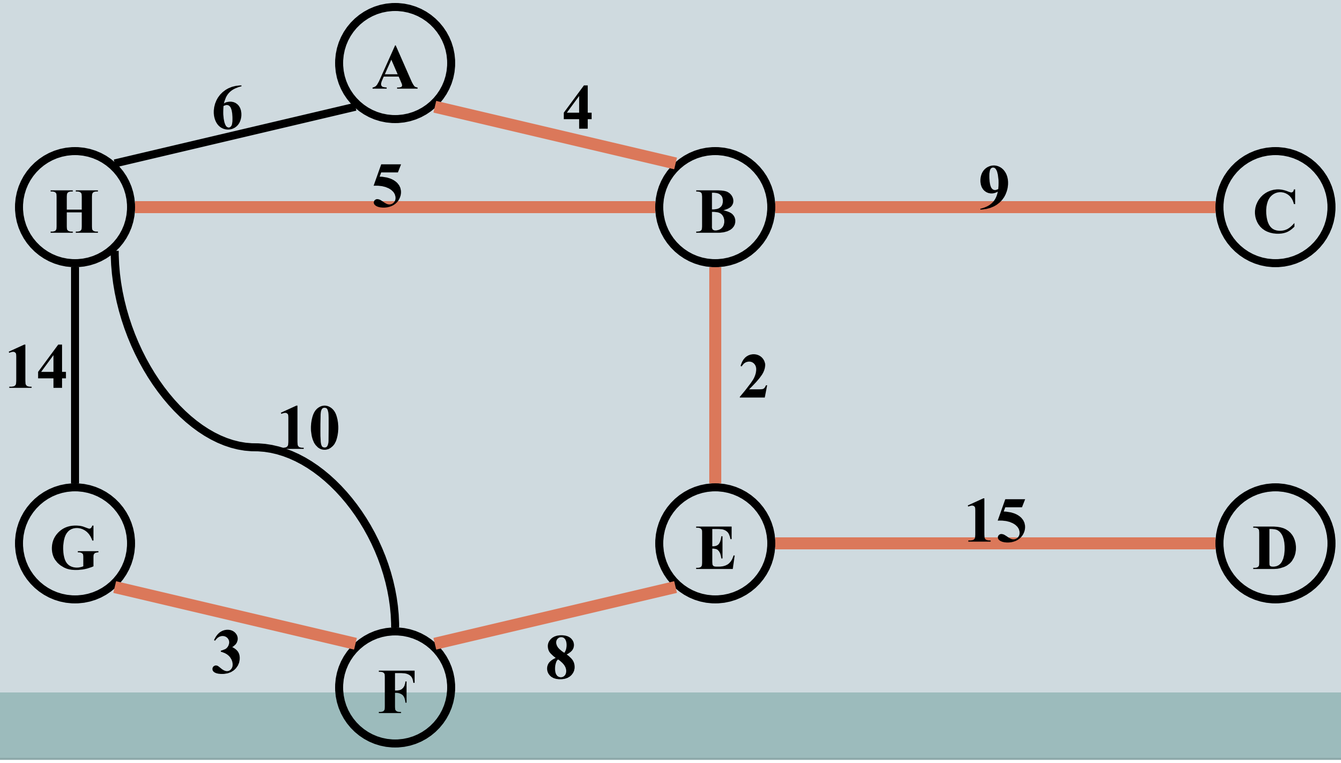
Minimum Spanning Tree

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



Minimum Spanning Tree

- Solution: (not unique usually..), edges?, cycles?



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)

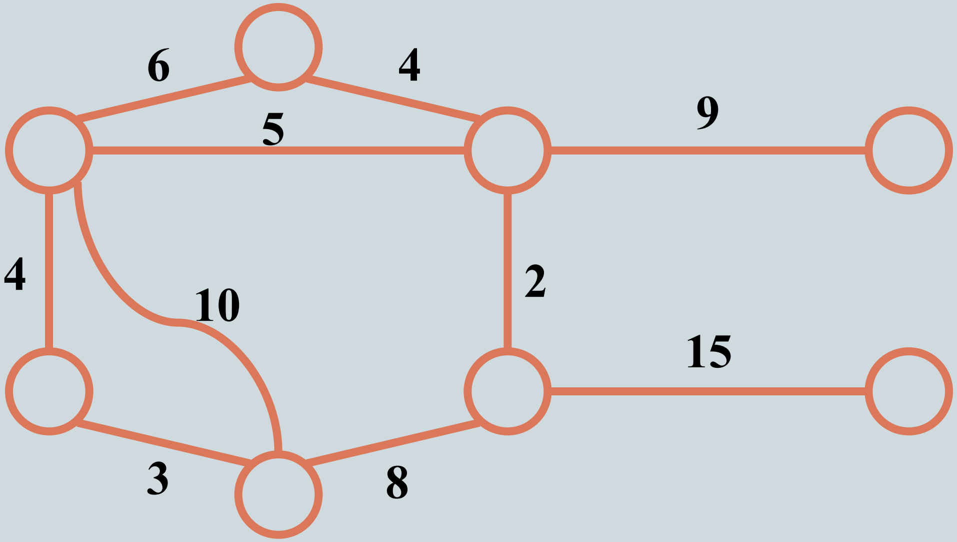
Minimum Spanning Tree

- 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
- Proof: in book (see Thm 23.1)

Prim’s algorithm (for MST)

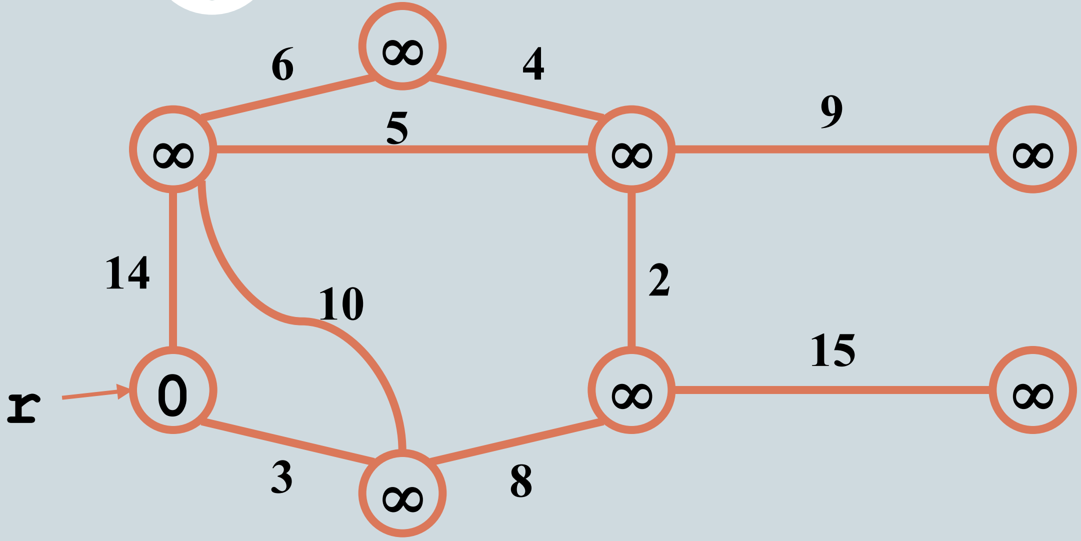
- Start at an arbitrary vertex and grow until the tree spans all the vertices
- In each step add to the tree an edge with minimum weight

Prim’s vs Kruskal’s Algorithm



Prim’s Algorithm

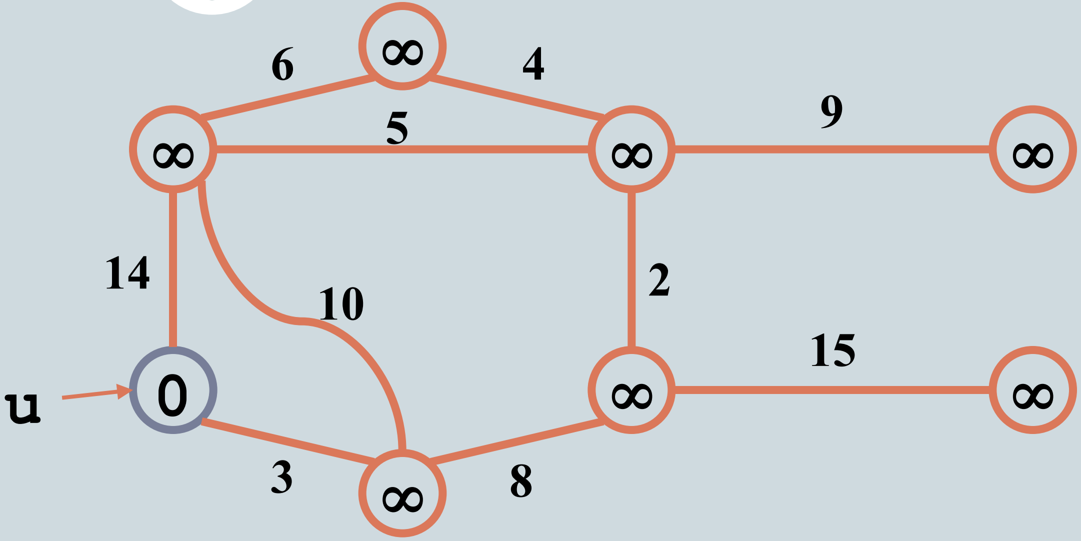
```
MST-Prim(G, w, r)
    Q = G.V;
    for each u ∈ Q
        u.key = ∞;
    r.key = 0;
    u.π = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < v.key)
                v.π = u;
                v.key = w(u,v);
```



Pick a start vertex r

Prim’s Algorithm

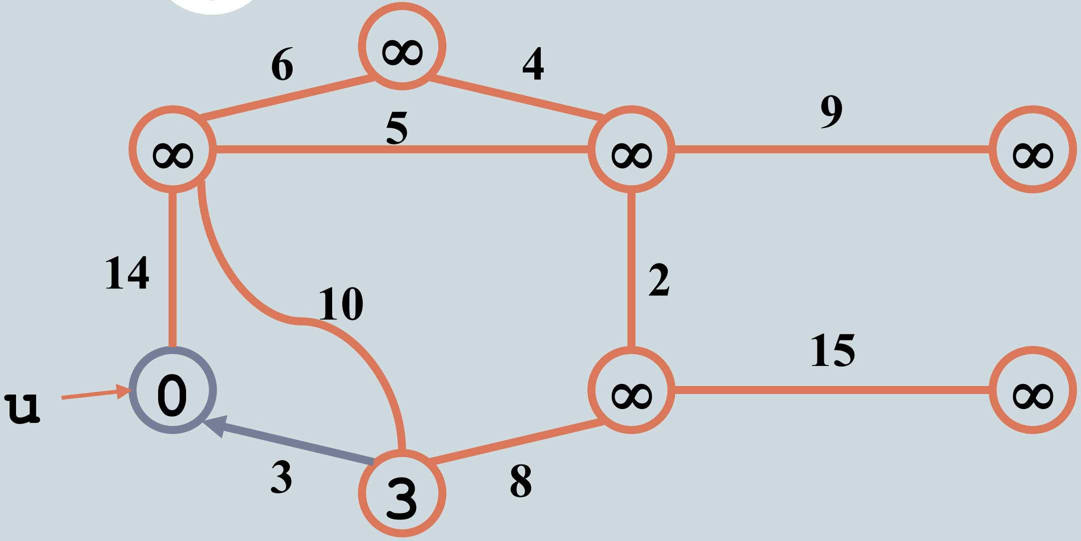
```
MST-Prim(G, w, r)
    Q = G.V;
    for each u ∈ Q
        u.key = ∞;
    r.key = 0;
    u.π = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < v.key)
                v.π = u;
                v.key = w(u,v);
```



Black vertices have been removed from Q

Prim’s Algorithm

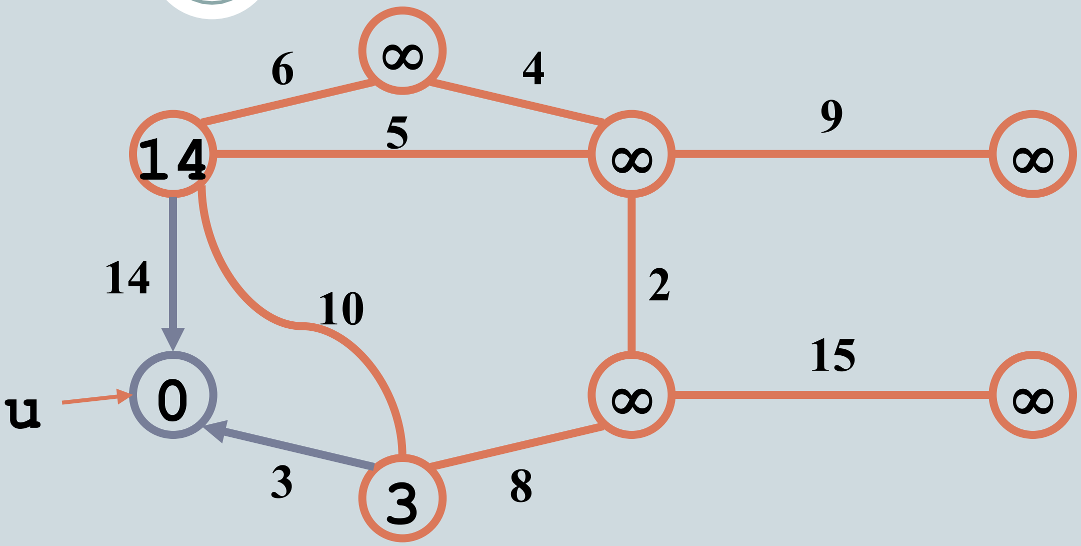
```
MST-Prim(G, w, r)
    Q = G.V;
    for each u ∈ Q
        u.key = ∞;
    r.key = 0;
    u.π = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < v.key)
                v.π = u;
                v.key = w(u,v);
```



Black arrows indicate parent pointers

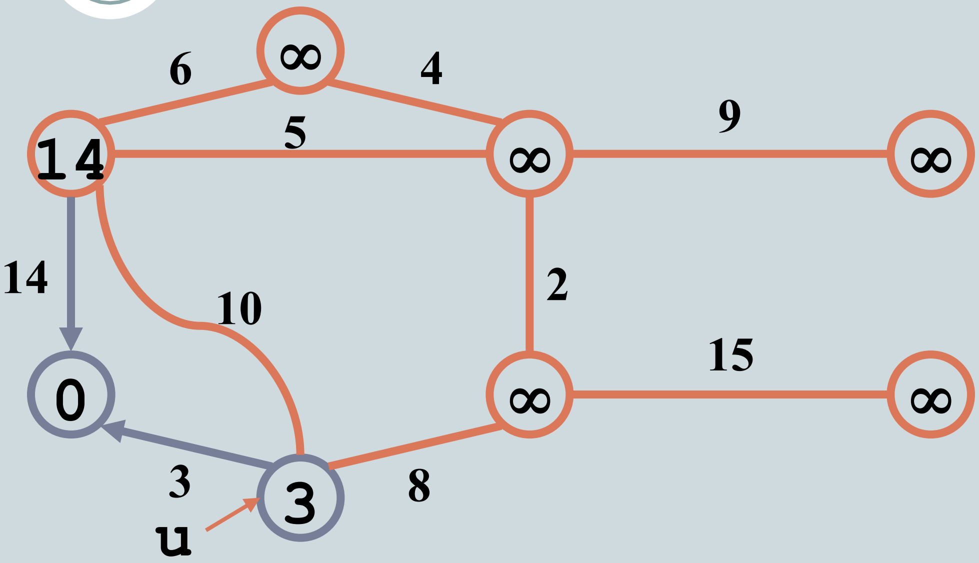
Prim’s Algorithm

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



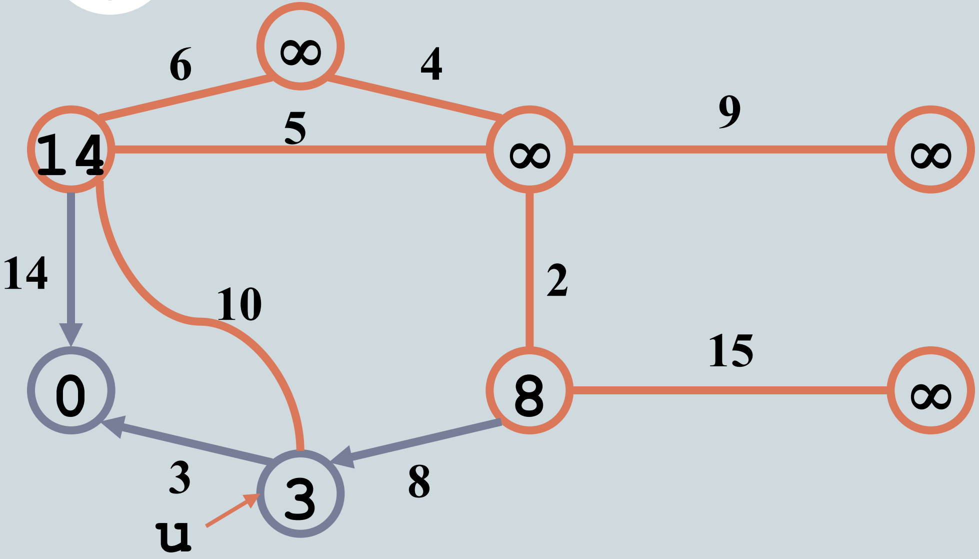
Prim’s Algorithm

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



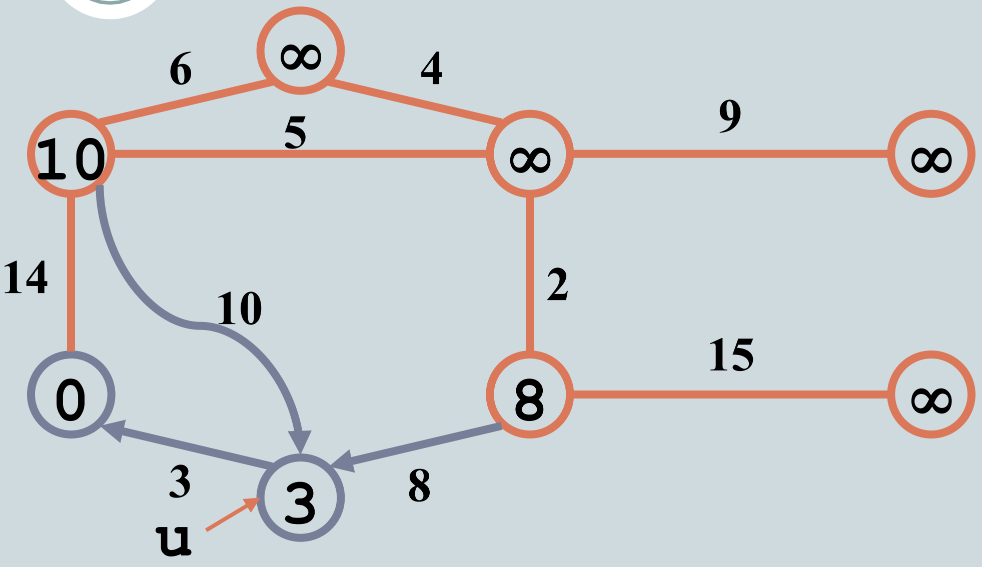
Prim's Algorithm

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



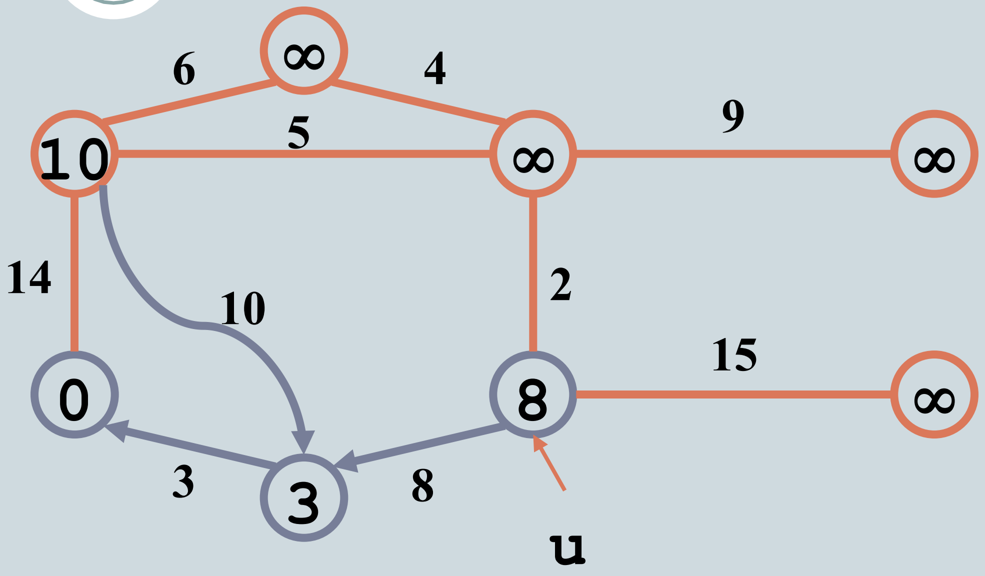
Prim’s Algorithm

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



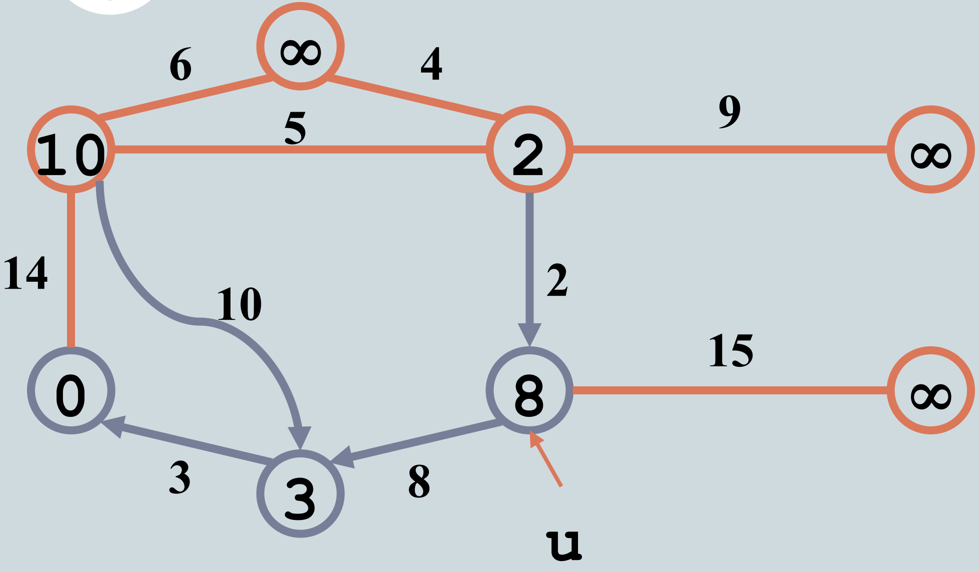
Prim’s Algorithm

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

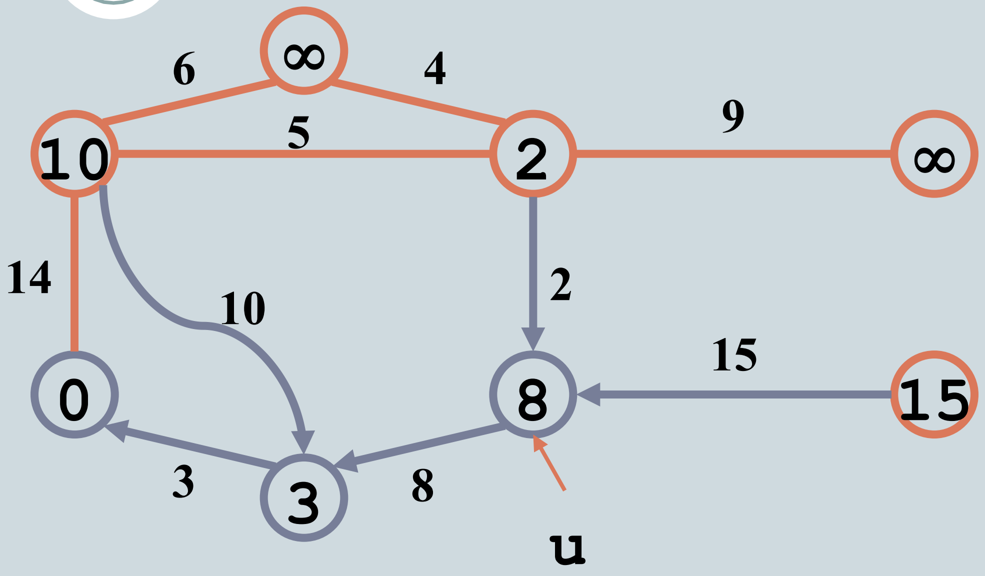


Prim’s Algorithm

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



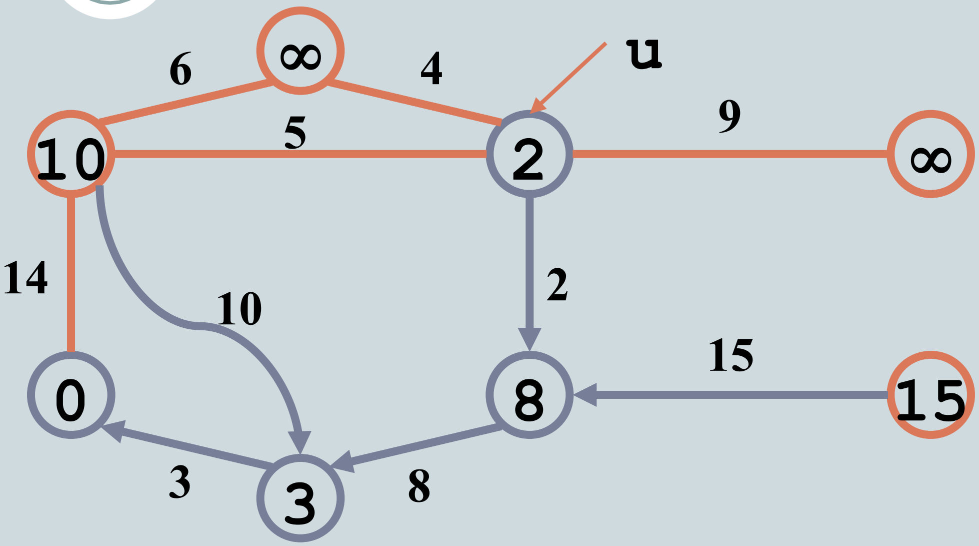
Prim’s Algorithm



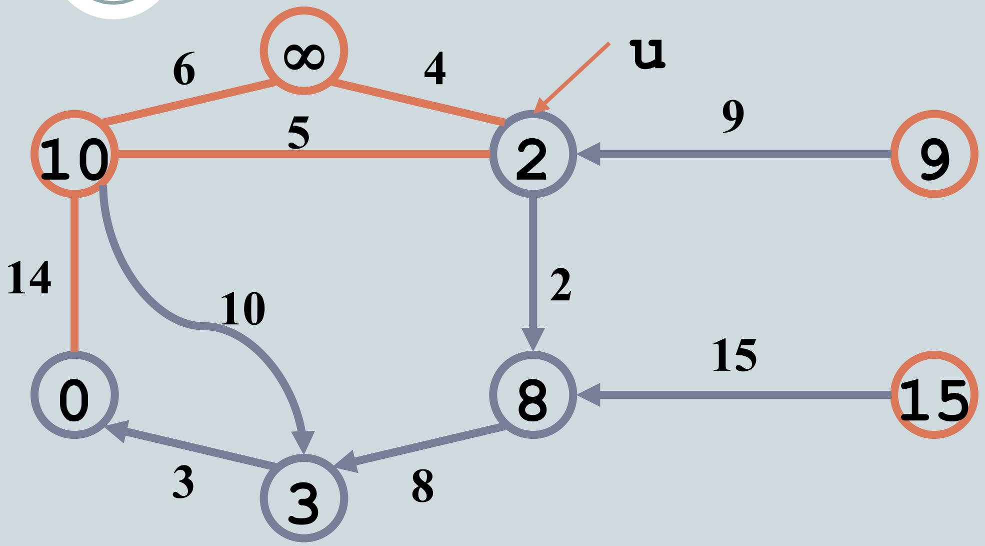
```
MST-Prim(G, w, r)
    Q = G.V;
    for each u ∈ Q
        u.key = ∞;
    r.key = 0;
    u.π = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < v.key)
                v.π = u;
                v.key = w(u,v);
```

Prim's Algorithm

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

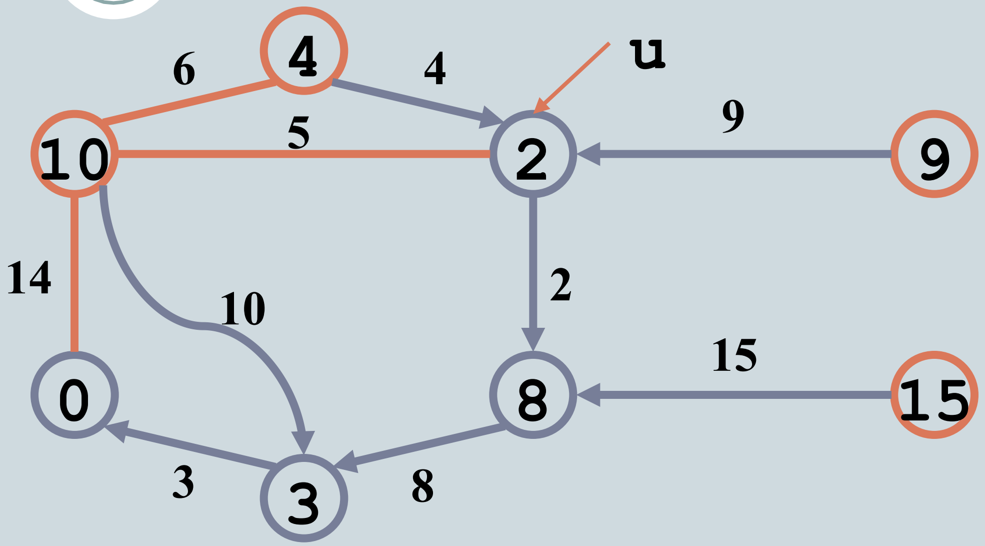


Prim's Algorithm



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

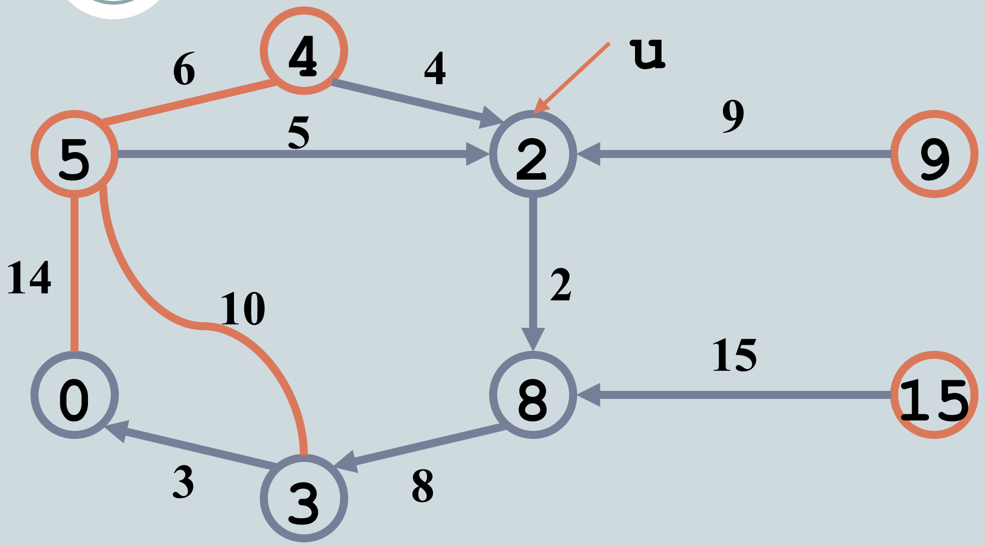
Prim’s Algorithm



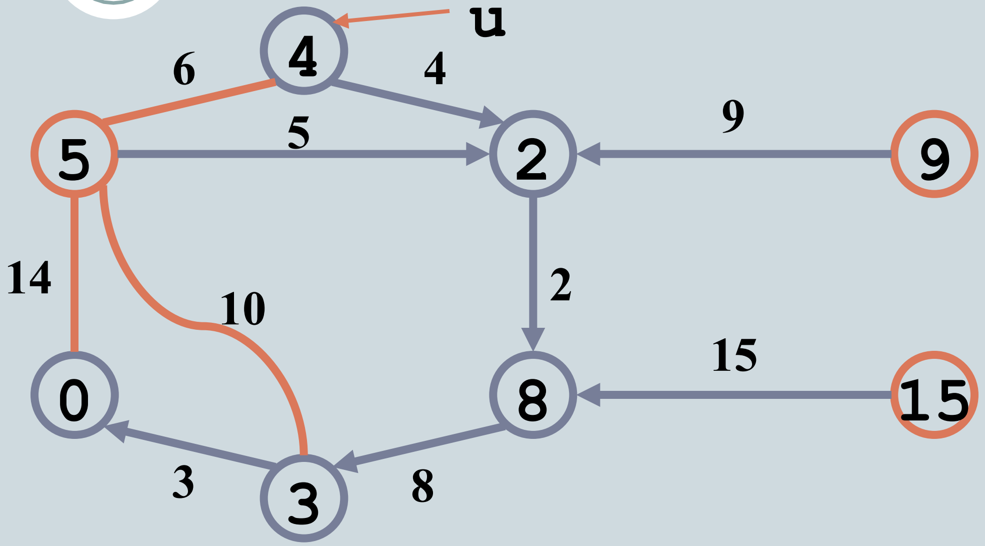
```
MST-Prim(G, w, r)
    Q = G.V;
    for each u ∈ Q
        u.key = ∞;
    r.key = 0;
    u.π = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < v.key)
                v.π = u;
                v.key = w(u,v);
```

Prim's Algorithm

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

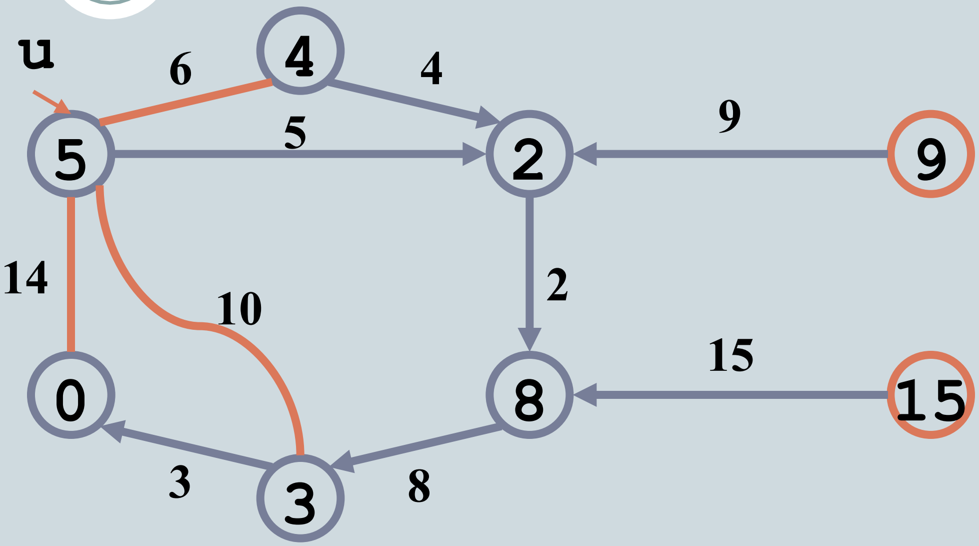


Prim’s Algorithm



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

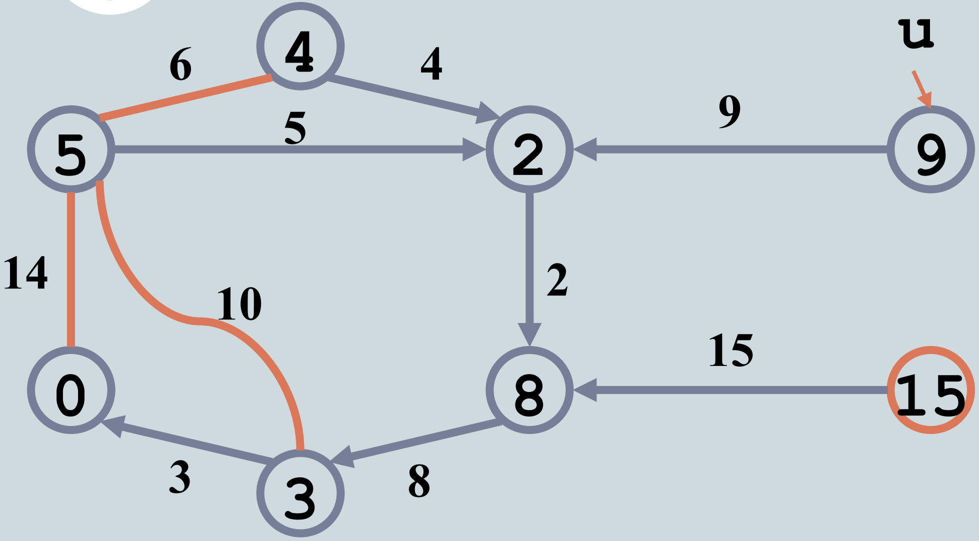
Prim’s Algorithm



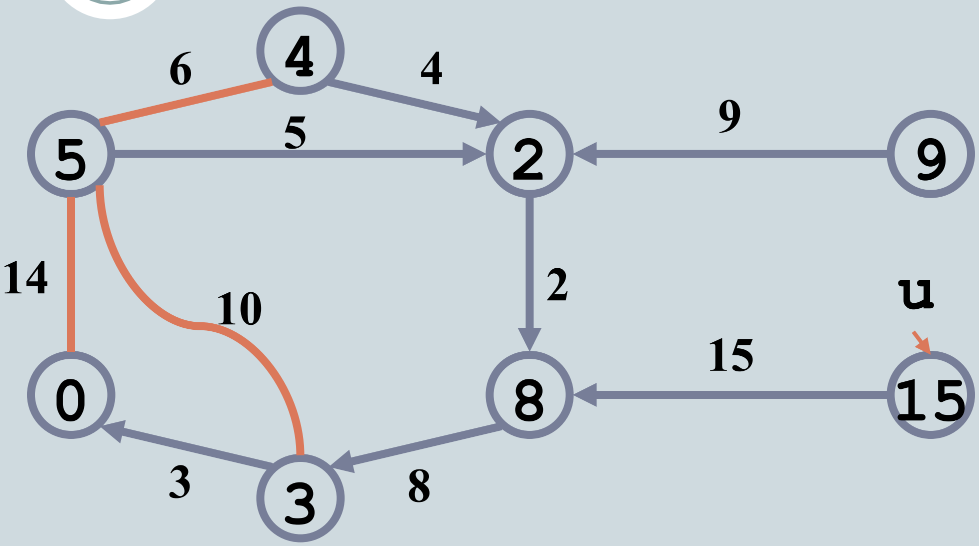
```
MST-Prim(G, w, r)
    Q = G.V;
    for each u ∈ Q
        u.key = ∞;
    r.key = 0;
    u.π = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < v.key)
                v.π = u;
                v.key = w(u,v);
```

Prim’s Algorithm

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

Review: Prim’s Algorithm

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

How much time to build heap Q?

|V|

How often is ExtractMin() called?

|V| each cost log|V|

How often is DecreaseKey() called?

|E| each cost log|V|

Review: Prim's Algorithm

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

Greedy Algorithms

- Used for optimization problems.
- ***Idea:*** When we have a choice to make
  - make the one that looks best *right now*.
  - make a *locally optimal choice* in hope of getting a *globally optimal solution.*

- Greedy algorithms don't always yield an optimal solution. But sometimes they do. We'll see a problem for which they do. Then we'll look at some general characteristics of when greedy algorithms give optimal solutions.
- Similar to dynamic programming (see later).

Kruskal’s Algorithm for MST

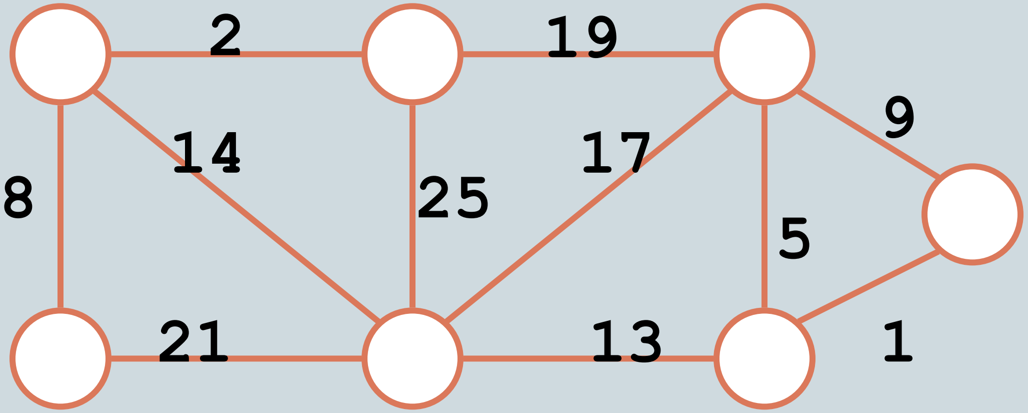
- Starts with each vertex being its own component.
- Repeatedly merges two components into one by choosing the light edge that connects them (i.e., the light edge crossing the cut between them).
- Scans the set of edges in monotonically increasing order by weight.
- Uses a disjoint-set data structure to determine whether an edge connects vertices in different components.

Kruskal's Algorithm

```
Kruskal()
{
    T = $\emptyset$;
    for each v $\in$ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) $\in$ E (in sorted order)
        if FindSet(u) $\neq$ FindSet(v)
            T = T $\cup$ {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

Kruskal's Algorithm

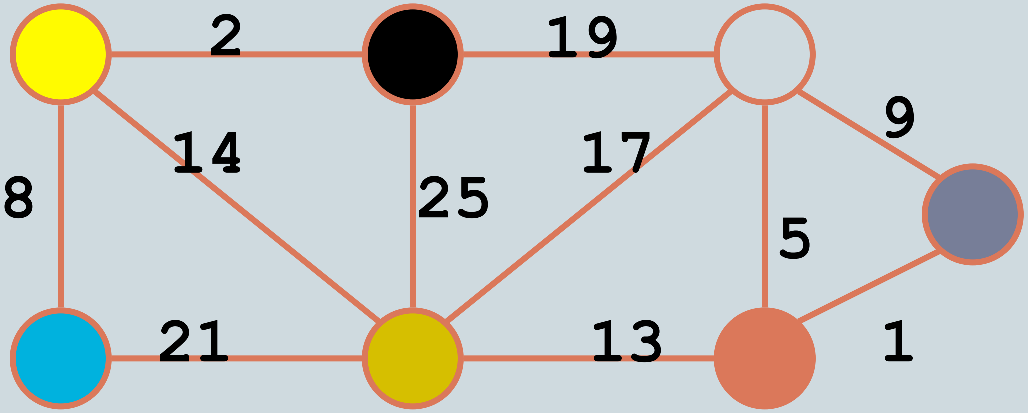
Run the algorithm:



```
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

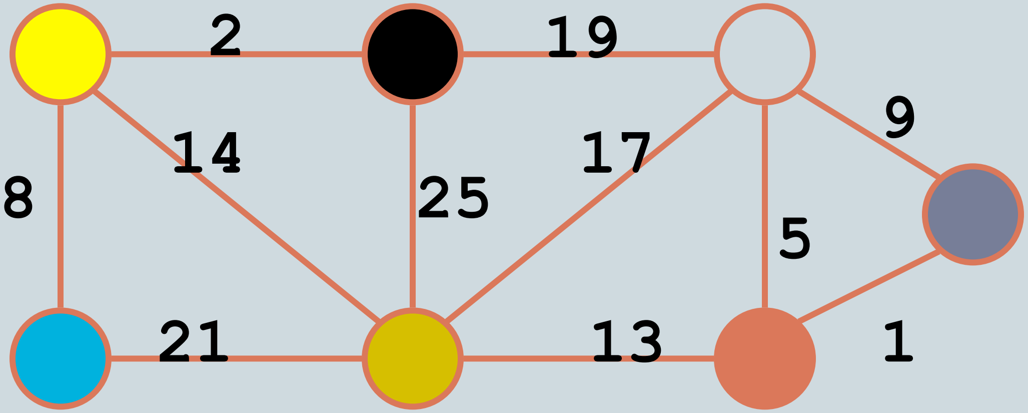
Run the algorithm:



```
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

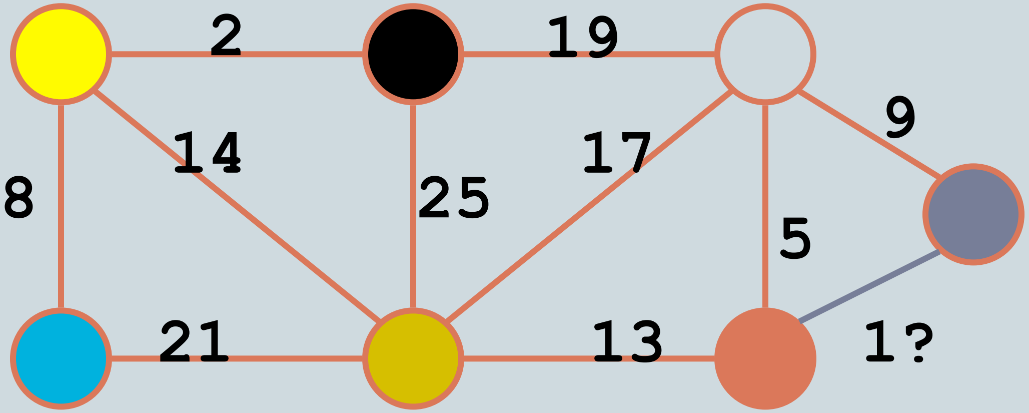
Run the algorithm:



```
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

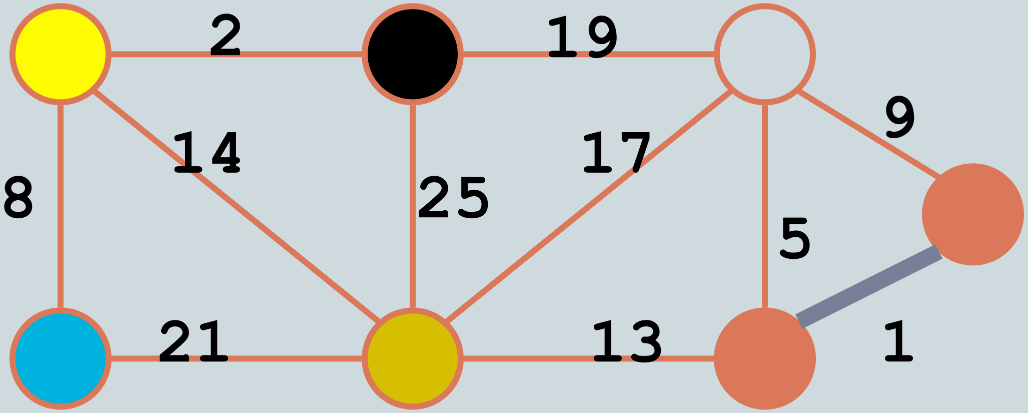
Run the algorithm:



```
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

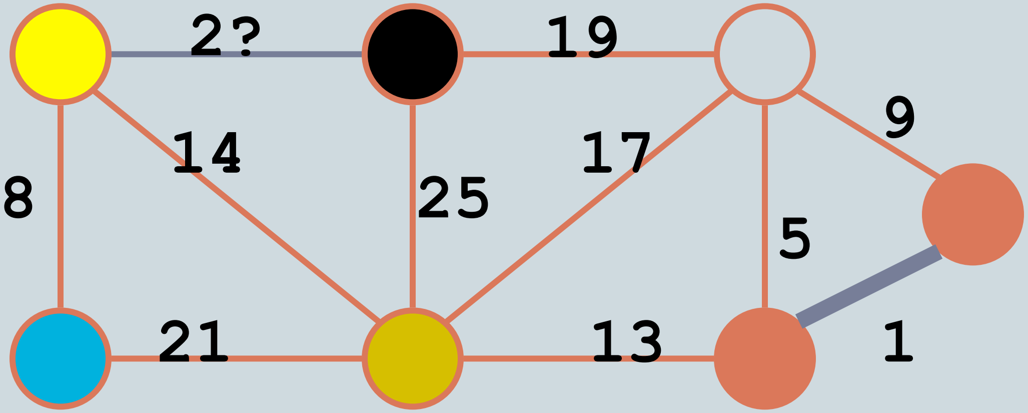
Run the algorithm:



```
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

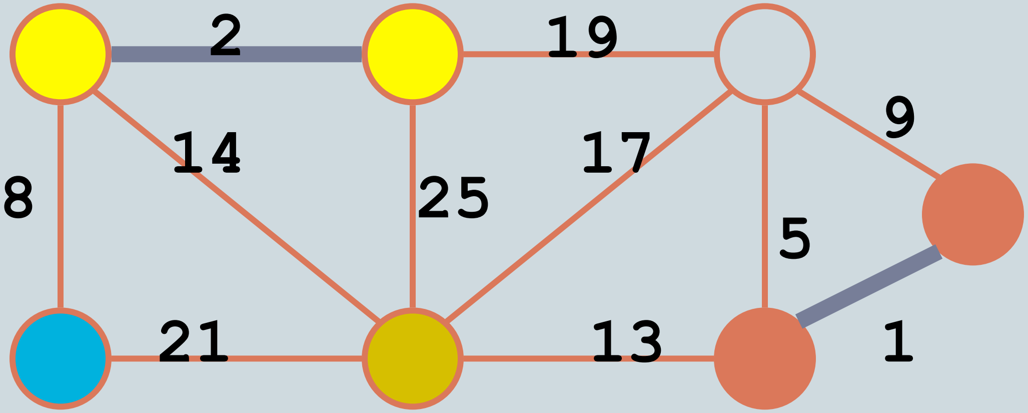
Run the algorithm:



```
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:

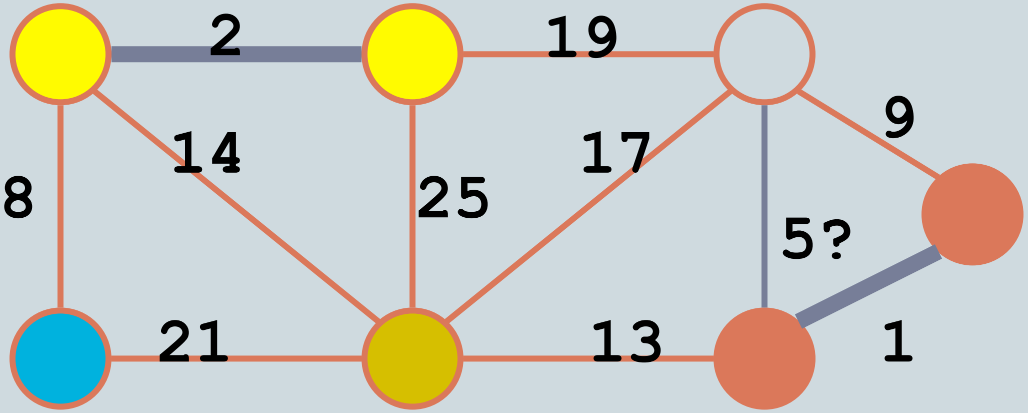


```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

Kruskal's Algorithm

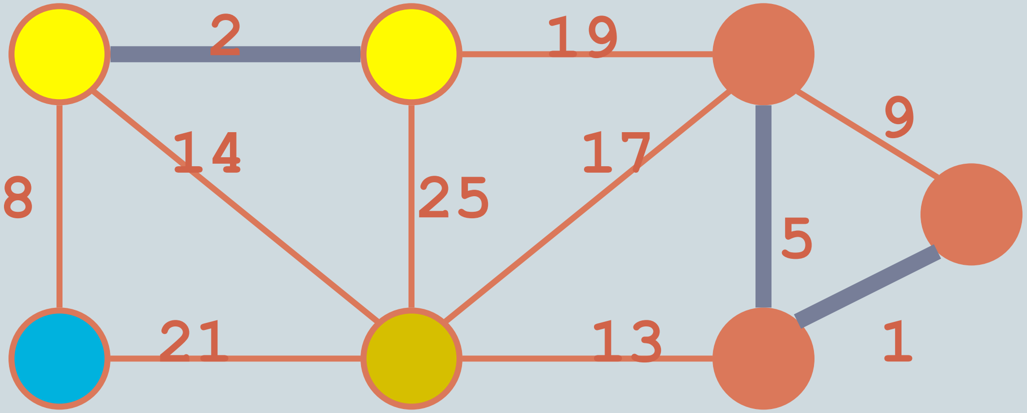
Run the algorithm:

```
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

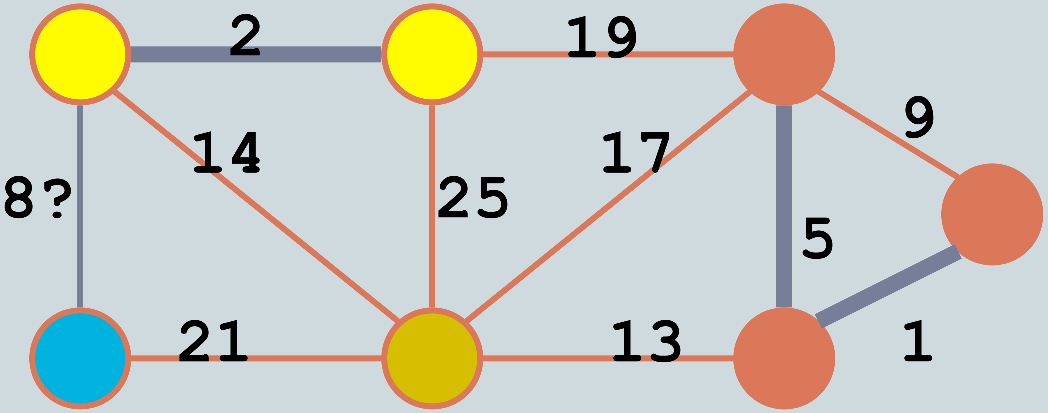
Run the algorithm:



```
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

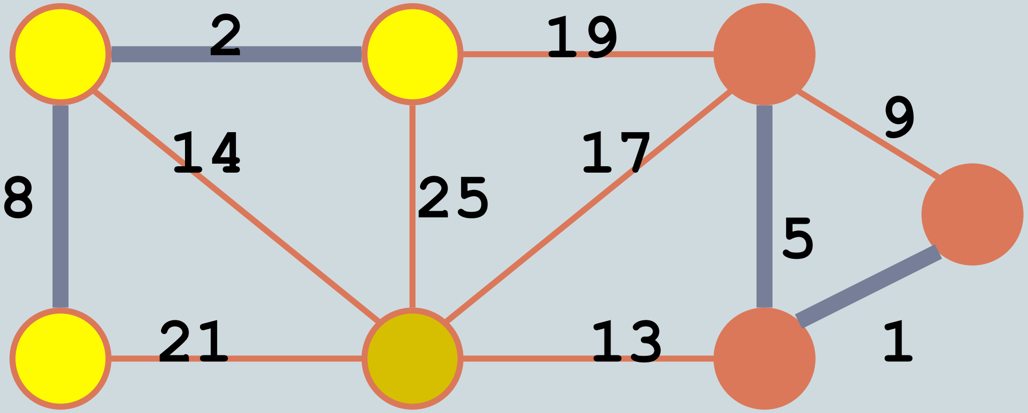
Run the algorithm:



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

Kruskal’s Algorithm

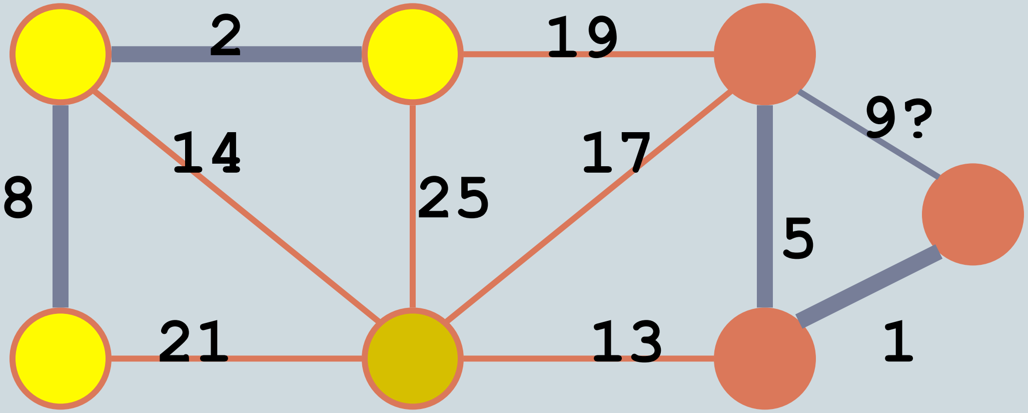
Run the algorithm:



```
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

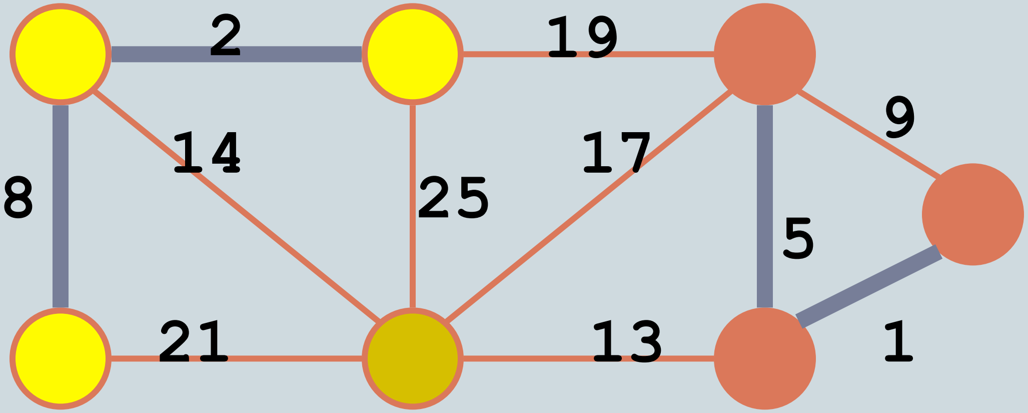
Run the algorithm:



```
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

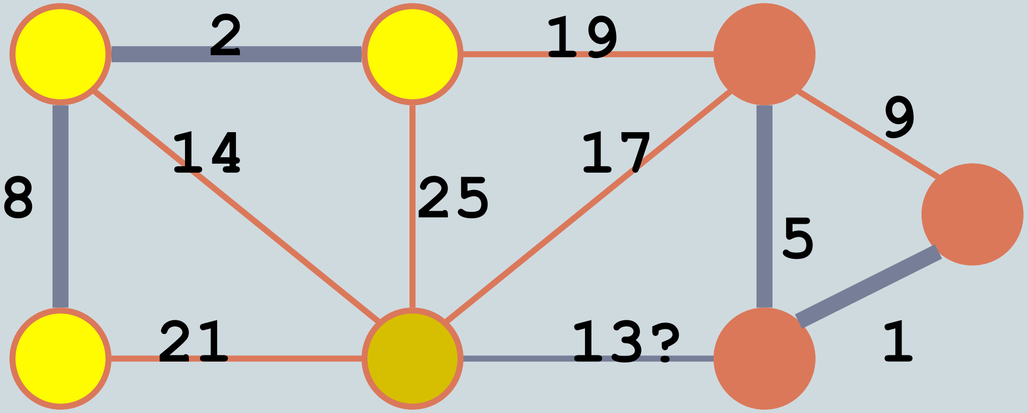
Run the algorithm:



```
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

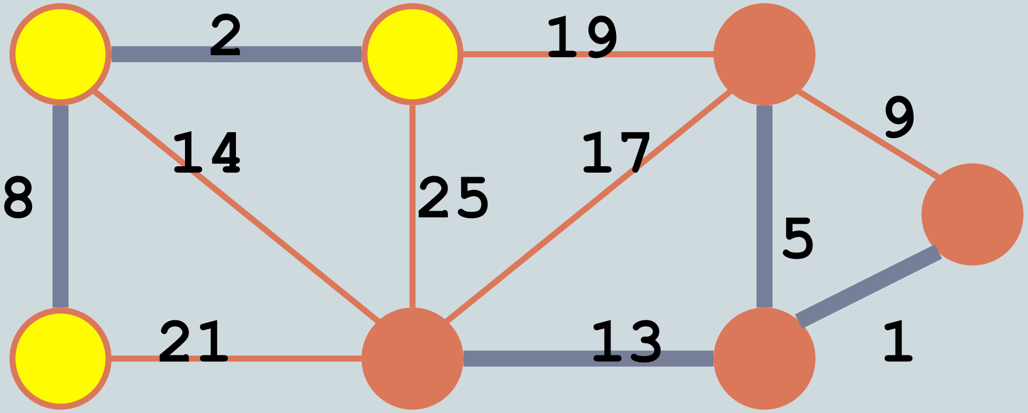
Run the algorithm:



```
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

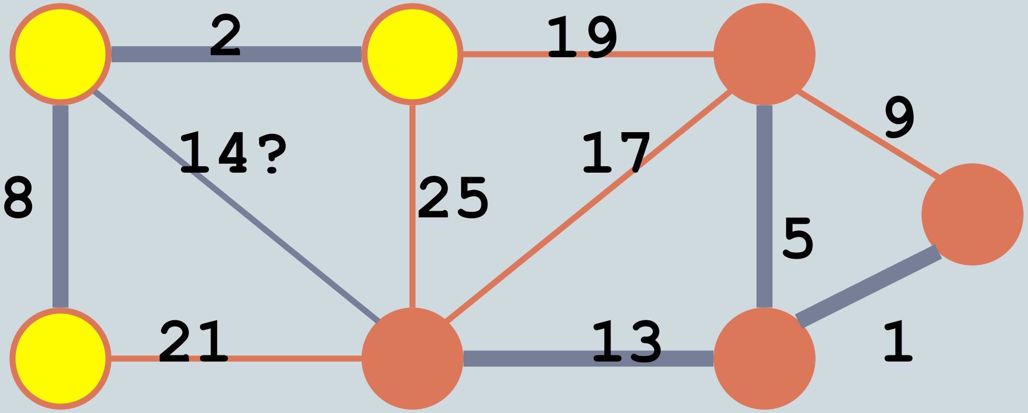
Run the algorithm:



```
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

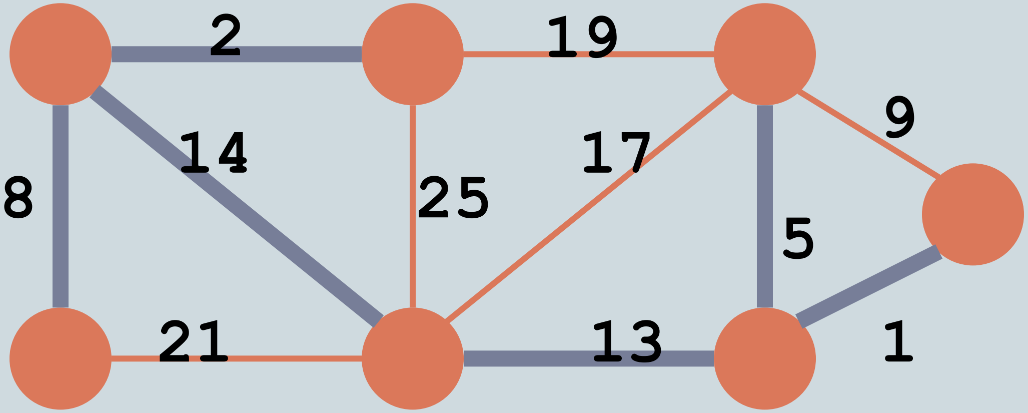
Run the algorithm:



```
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

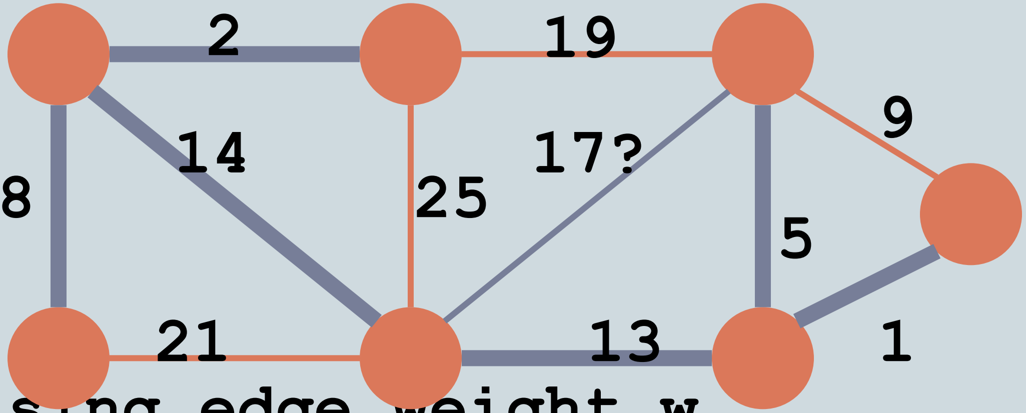
Run the algorithm:



```
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

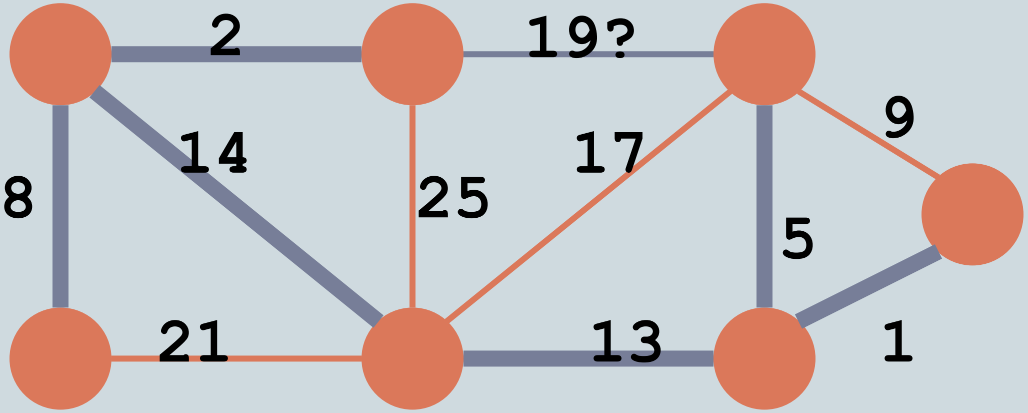
Run the algorithm:



```
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

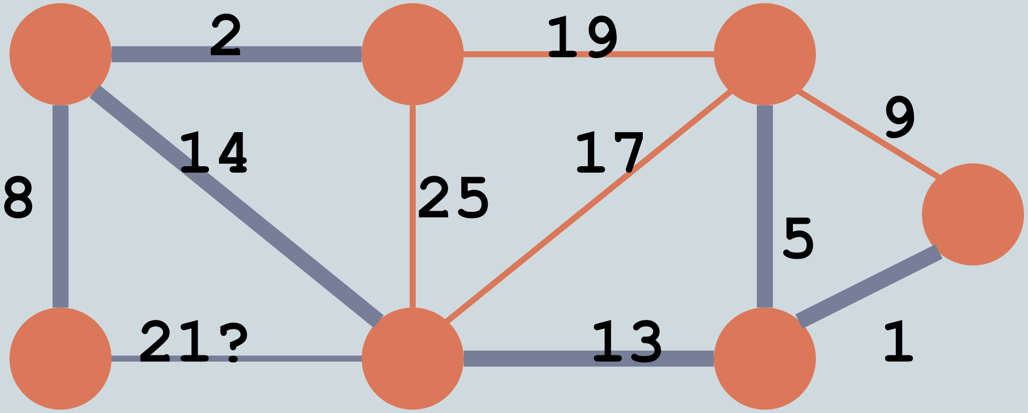
Run the algorithm:



```
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

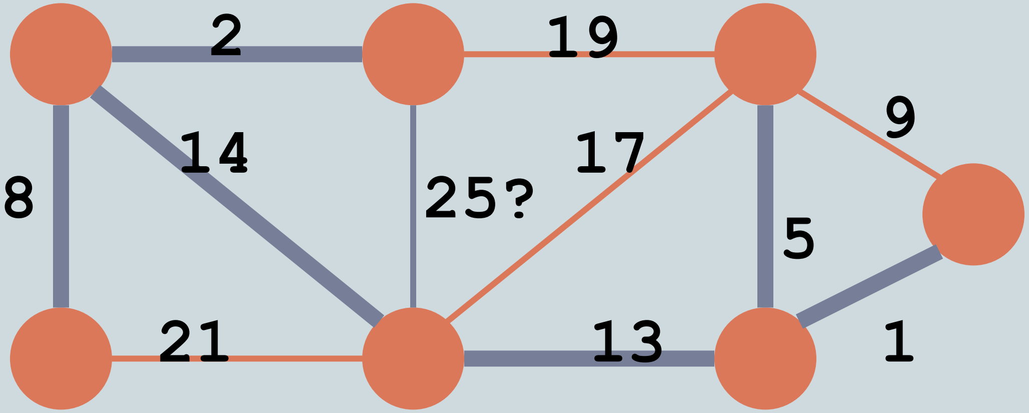
Run the algorithm:



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

Kruskal’s Algorithm

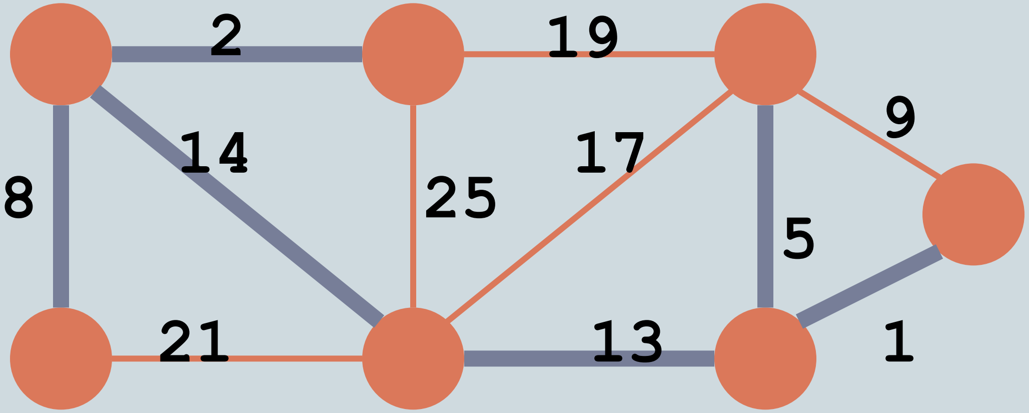
Run the algorithm:



```
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

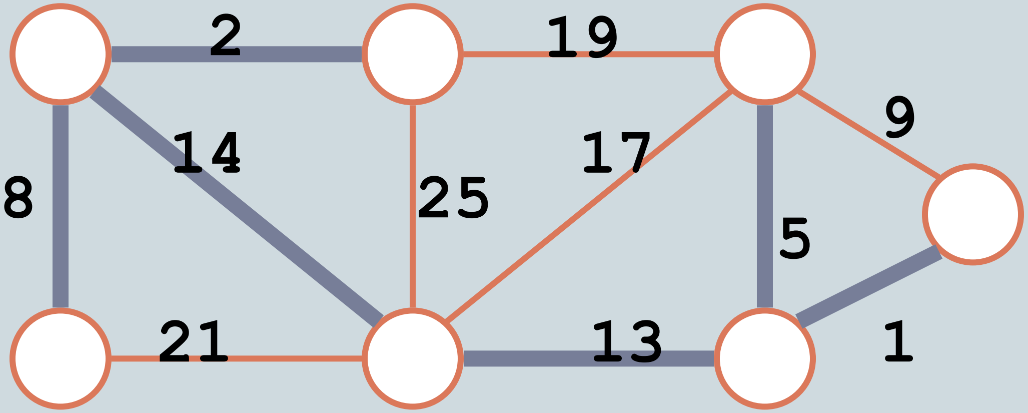
Run the algorithm:



```
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,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));
}
```

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

Kruskal’s Algorithm

What will affect the running time?

```
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

What will affect the running time?
O(ElgE) Sort Edges
O(V) MakeSet() calls
O(E) FindSet() calls
O(V) Union() calls
(Exactly how many Union()s?)

```
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: Running Time

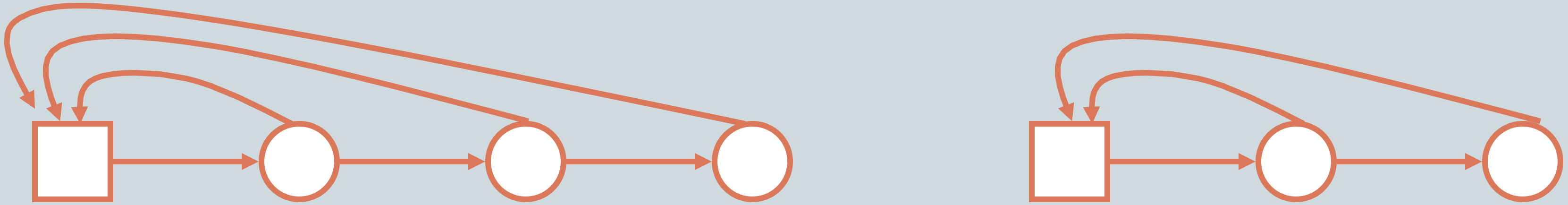
- To summarize:
  - Sort edges: O(E lg E)
  - O(V) MakeSet()'s
  - O(E) FindSet()'s
  - O(V) Union()'s
- So:
  - Best disjoint-set union algorithm makes above 3 operations take $O(E \cdot \alpha(E,V))$, α almost constant
  - Overall thus O(E lg E), almost linear w/o sorting

Disjoint-Set Union Problem (Ch 21)

- Want a data structure to support disjoint sets
  - Collection of disjoint sets $\mathcal{S} = \{S_i\}$, $S_i \cap S_j = \varnothing$
- Need to support following operations:
  - MakeSet(x): $\mathcal{S} = \mathcal{S} \cup \{\{x\}\}$
  - Union(S_i, S_j): $\mathcal{S} = \mathcal{S} - \{S_i, S_j\} \cup \{S_i \cup S_j\}$
  - FindSet(X): return $S_i \in \mathcal{S}$ such that $x \in S_i$
- Application: MST (Kruskal's algorithm)

Disjoint Set Union (Ch 21.2)

- So how do we implement disjoint-set union?
  - Naïve implementation: use a linked list to represent each set:



    - MakeSet(): O(1) time
    - FindSet(): O(1) time
    - Union(A,B): “copy” elements of A into B: O(A) time
  - *How long can a single Union() take?*
  - *How long will n Union()’s take?*

Disjoint Set Union: Analysis

- Worst-case analysis: O(n^2) time for n Union’s

  Union(S_1, S_2) “copy” 1 element

  Union(S_2, S_3) “copy” 2 elements

  …

  Union(S_{n-1}, S_n) “copy” n-1 elements

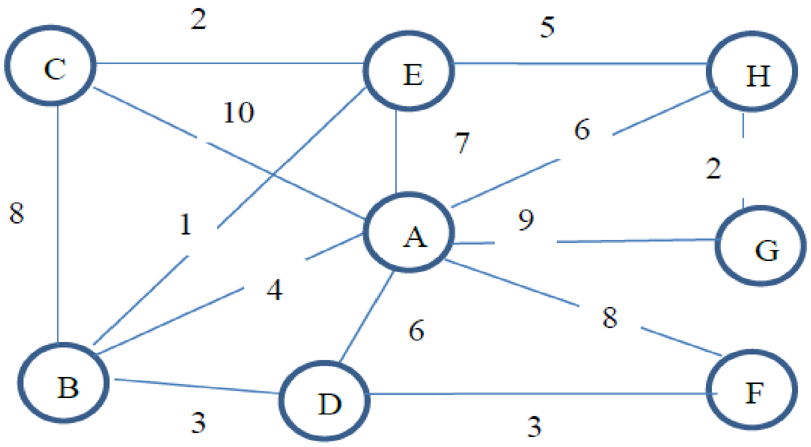
  O(n^2)

- Improvement: always copy smaller into larger
  - *Why will this make things better?*
  - *What is the worst-case time of Union()?*
- But now n Union’s take only O(n lg n) time!

practice

points each]

13. **[18 pts]** LuxuryForAll Construction is in the process of installing power lines to a large housing development. The owner wants to minimize the total length of wire used, which will minimize her costs. The housing development is shown as a graph in the next figure. Each house has been numbered, and the distance between the houses are given in hundreds of feet. What do you recommend? (Total length of wires and also which will be installed.) How would you solve it efficiently? Show all the steps.



Next: Single-Source Shortest Path (Ch 24)

- Input: given a weighted directed graph G,
- Output: find the path from a given source vertex s to another vertex v
- Goal: minimum-weight path

  - "Shortest-path" = minimum weight
  - Weight of path is sum of weighted edges
  - E.g., a road map: what is the shortest path from San Jose to Palo Alto?