# COT 6405 Introduction to Theory of Algorithms

## Topic 15. Minimum Spanning Tree

# Minimum Spanning Tree

- Problem:
  - given a connected, undirected, weighted graph G = (V, E)
  - find a spanning tree using edges that connects all nodes with a minimal total weight w(T)= SUM(w[u,v])
    - w[u,v] is the weight of edge (u,v)
- Objectives: we will learn
  - Generic MST
  - Kruskal's algorithm
  - Prim's algorithm

# Motivation Example

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

# Model as a graph

- The problem can be modeled as a graph
  - Undirected weighted graph *G = (V, E)*.
  - Weight *w(u, v)* on each edge *(u, v)* ∈ *E*.
- Find $T \subseteq E$ , such that
  - *T* connects all vertices (*T* is a spanning tree)
  - $w(T) = \sum_{(u,v)\in T} w(u,v)$ is minimized.
- A spanning tree whose weight is minimum over all spanning trees is called a minimum spanning tree, *MST*.

# Growing a minimum spanning tree

- Building up the solution
  - We will build a set *A* of edges
  - Initially, *A* has no edges.
  - As we add edges to *A*, maintain a loop invariant
- Loop invariant: *A* is a subset of some MST
  - Add only edges that maintain the invariant
  - Definition: If *A* is a subset of some MST, an edge *(u, v)* is safe for *A*, if and only if $A \cup \{(u, v)\}$ is also a subset of some MST
  - So we will add only safe edges

# Generic MST algorithm

GENERIC-MST$(G, w)$

$A = \emptyset$
while $A$ is not a spanning tree
    find an edge $(u, v)$ that is safe for $A$
    $A = A \cup \{(u, v)\}$
return $A$

# Correctness

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

# Definitions

- Let $S \subset V$ (vertex set); $A \subseteq E$ (edge set).
- A cut $(S, V - S)$ is a partition of vertices into two disjoint sets: S and V-S
- Edge $(u, v) \in E$ **crosses** the cut $(S, V-S)$ if one endpoint is in S and the other is in $V-S$.
- A cut **respects** edge set A, if and only if no edge in A crosses the cut.
- An edge is a **light edge** crossing a cut, if and only if its weight is minimum over all edges crossing the cut.
  - For a given cut, there can be > 1 light edge crossing it.

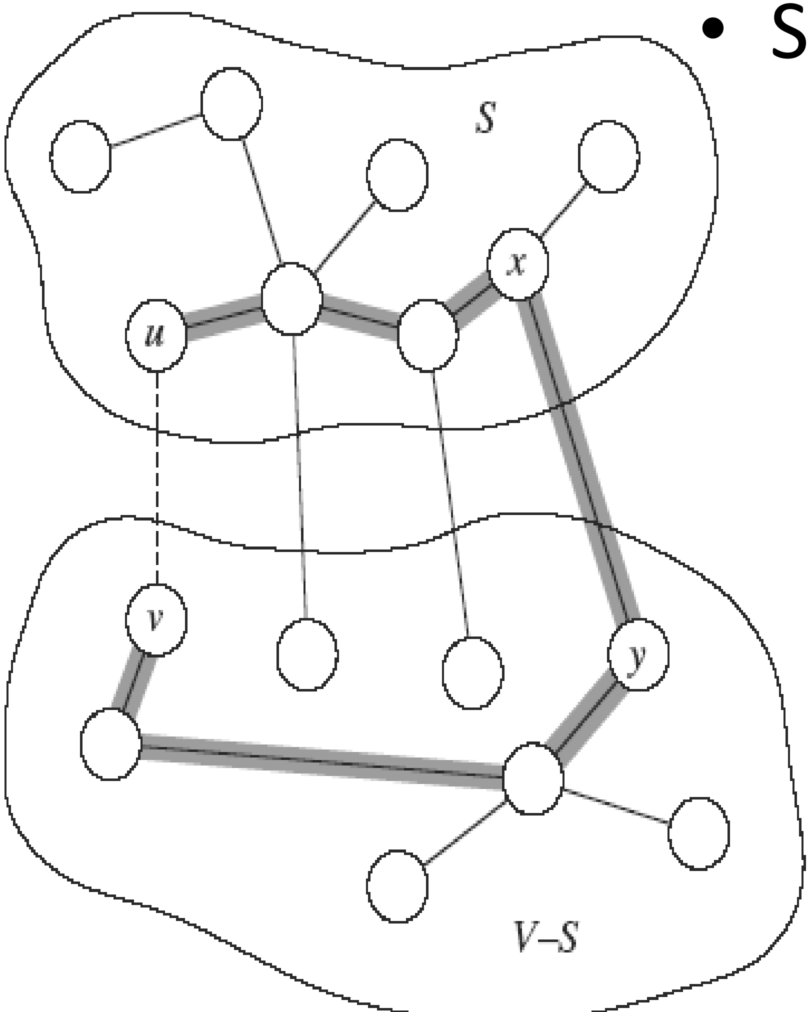
# Theorem

- Let edge set *A* be a subset of some MST
- *(S, V −S)* be a cut that respects edge set *A*
  - No edges in A crosses the cut
- *(u, v)* be a light edge crossing cut *(S, V −S)*.
- Then, *(u, v)* is safe for *A*.
- **Proof**
  - Let tree T be an MST that includes edge set *A*
  - If *T* contains edge *(u, v)*, done.
  - So, now assume that *T* does not contain edge *(u, v)*
  - We'll construct a different MST T' that includes *A* ∪ {*(u, v)*}.

# Proof

- Recall: a tree has a unique path between each pair of vertices (why?).
  - Since *T* is an MST, it contains a unique path *p* between *u* and *v*.
  - Path *p* must cross the cut *(S, V–S)* once
  - Let *(x, y)* be an edge of *p* that crosses the cut
- As *(u,v)* is a light edge, we have w(u,v) ≤ w(x,y)
- Since the cut respects *A*, edge *(x, y)* is not in *A*
- We can build tree T ’ from *T*
  - Remove *(x, y)*: Breaks *T* into two components.
  - Reconnects them with edge *(u,v)* → *T’*

# *Proof*




- Except for the dashed edge *(u, v)* , all edges shown are in *T*
- Shaded edges are the path *p*

# *Proof*

- So $T' = T - \{(x, y)\} \cup \{(u, v)\}$.
- → T' is another spanning tree
- $w(T') = w(T) - w(x, y) + w(u, v) \leq w(T)$
  - since $w(u, v) \leq w(x, y)$
  - Since (1) T' is a spanning tree, (2) $w(T') \leq w(T)$, and (3) T is an MST → T' must be an MST
- Need to show that $A \cup \{(u, v)\} \subset T'$
  - $A \subseteq T$ and $(x, y) \notin A$ $\Rightarrow A \subseteq T - \{(x, y)\}$
  - $A \cup \{(u, v)\} \subseteq T - \{(x, y)\} \cup \{(u, v)\} = T'$
  - Since *T'* is an MST, edge *(u, v)* is safe for *A*.

# MST: optimal substructure

- 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 $G_2 = (V_2,E_2)$
- 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)

# Corollary

- If $C = (V_C, E_C)$ is a connected component in the forest $G_A = (V, A)$
- $(u, v)$ is a light edge connecting $C$ to some other component in $G_A$
  - i.e.,$(u,v)$ is a light edge crossing the cut $(V_C,V-V_C)$
- Then, edge $(u, v)$ is safe for $A$.
- **Proof:** Set $S = V_C$ in the theorem.
  - This naturally leads to the Kruskal's algorithm

# Kruskal's algorithm

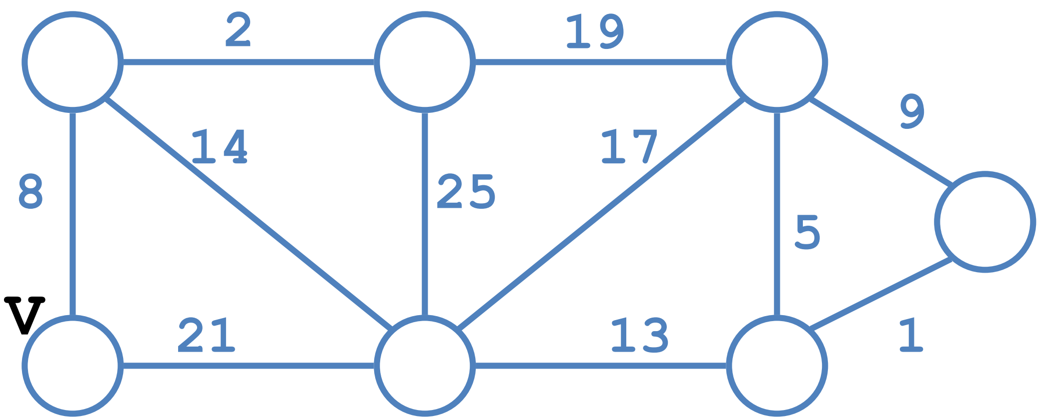
- Starts with each vertex being its own component
- Repeatedly merges two components into one by choosing the light edge that connects them
- Scans the set of edges in monotonically increasing order by weight
- Uses a disjoint-set data structure to determine whether an edge connects vertices in different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

# Disjoint Sets Data Structure

- A disjoint-set is a collection C ={$S_1$, $S_2$,…, $S_k$} of distinct dynamic sets
- Each set is identified by a member of the set, called representative.
- Disjoint set operations:
  - MAKE-SET(x): create a new set with only x
    - assume x is not already in some other set.
  - UNION(x,y): combine the two sets containing x and y into one new set.
    - A new representative is selected.
  - FIND-SET(x): return the representative of the set containing x.

# Kruskal's Algorithm

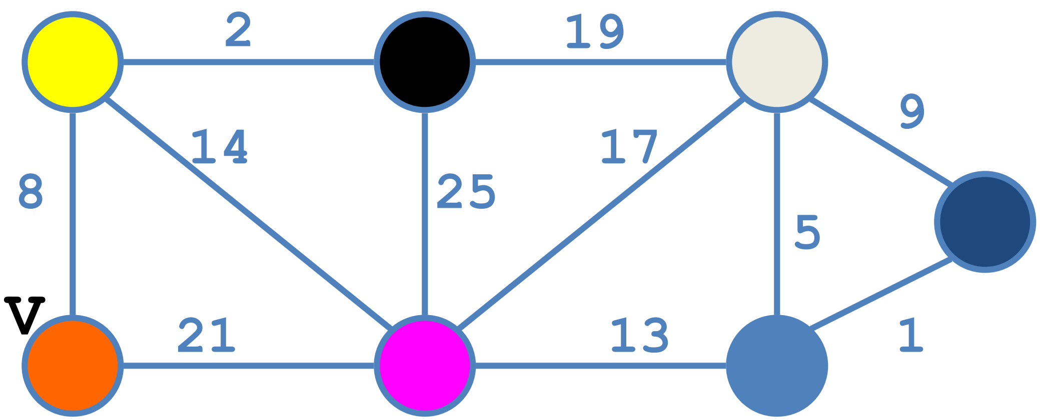
*Run the algorithm:*




```
Kruskal(G, w)
{
    A = ∅;
    for each v ∈ G.V
        Make-Set(v);
    sort G.E by non-decreasing order by weight w
    for each (u,v) ∈ G.E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            A = A U {{u,v}};
            Union(u, v);
}
```

# Kruskal's Algorithm

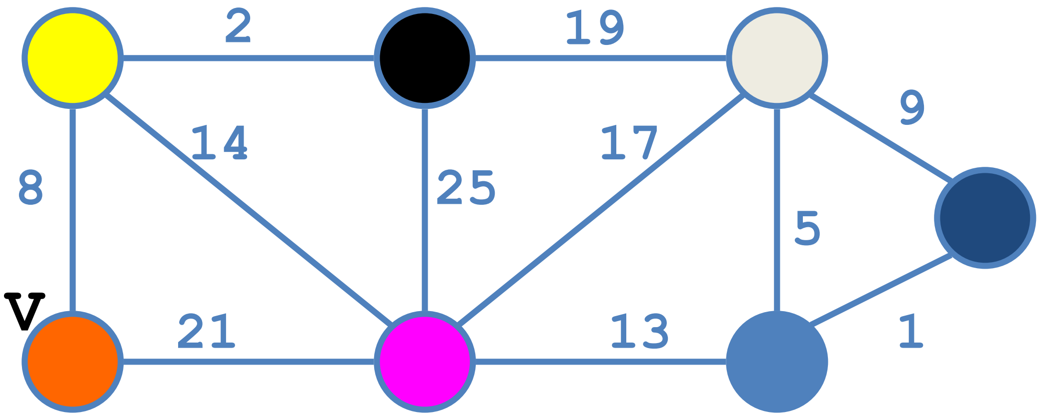
*Run the algorithm:*




```
Kruskal(G, w)
{
   A = ∅;
   for each v ∈ G.V
      Make-Set(v);
   sort G.E by non-decreasing order by weight w
   for each (u,v) ∈ G.E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         A = A U {{u,v}};
         Union(u, v);
}
```

# Kruskal’s Algorithm

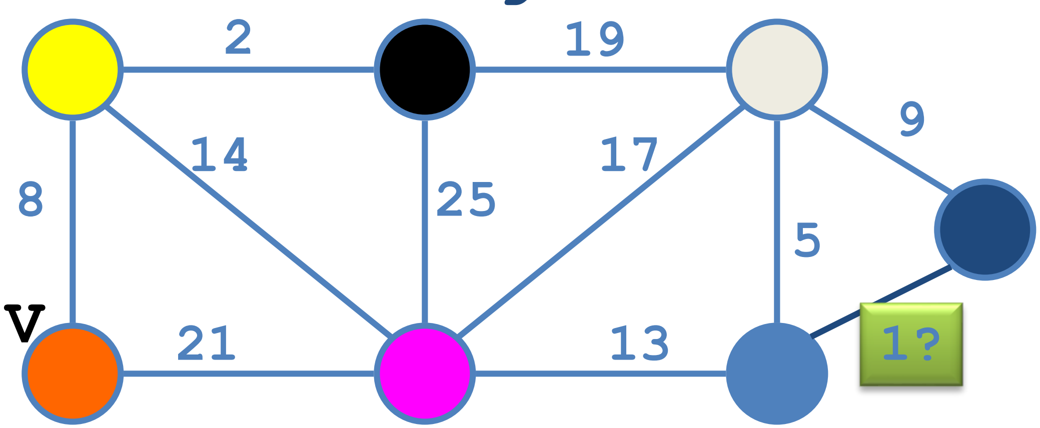
***Run the algorithm:***




```
Kruskal(G, w)
{
    A = ∅;
    for each v ∈ G.V
        Make-Set(v);
  { sort G.E by non-decreasing order by weight w
    for each (u,v) ∈ G.E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            A = A ∪ {{u,v}};
            Union(u, v);
}
```

# Kruskal's Algorithm

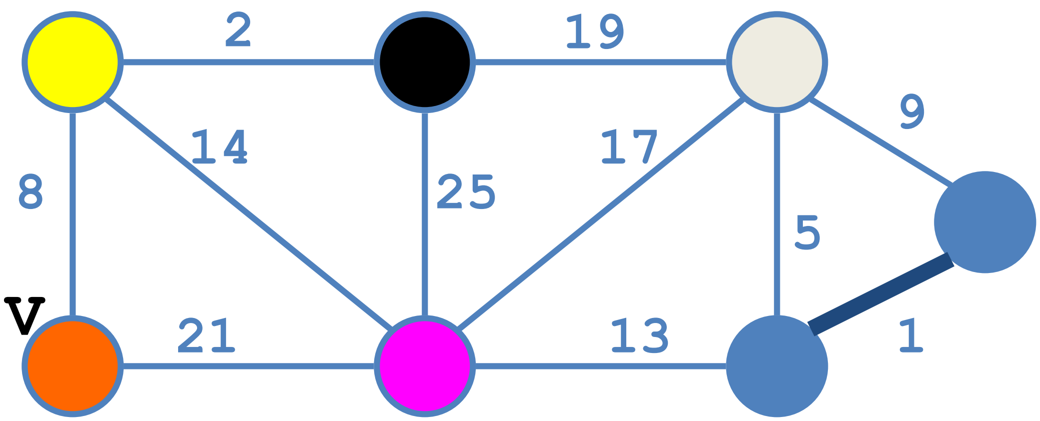
*Run the algorithm:*




```
Kruskal(G, w)
{
    A = ∅;
    for each v ∈ G.V
        Make-Set(v);
    sort G.E by non-decreasing order by weight w
    for each (u,v) ∈ G.E (in sorted order)
        if FindSet(u) ≠ FindSet(v) // same tree?
            A = A U {{u,v}};
            Union(u, v);
}
```

# Kruskal's Algorithm

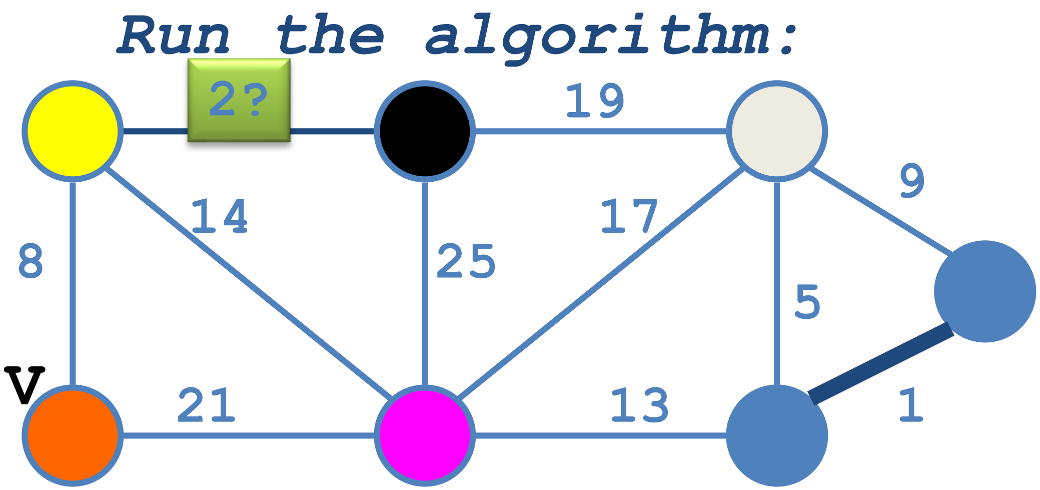
***Run the algorithm:***




```
Kruskal(G, w)
{
    A = ∅;
    for each v ∈ G.V
        Make-Set(v);
    sort G.E by non-decreasing order by weight w
    for each (u,v) ∈ G.E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            A = A U {{u,v}};
            Union(u, v);
}
```

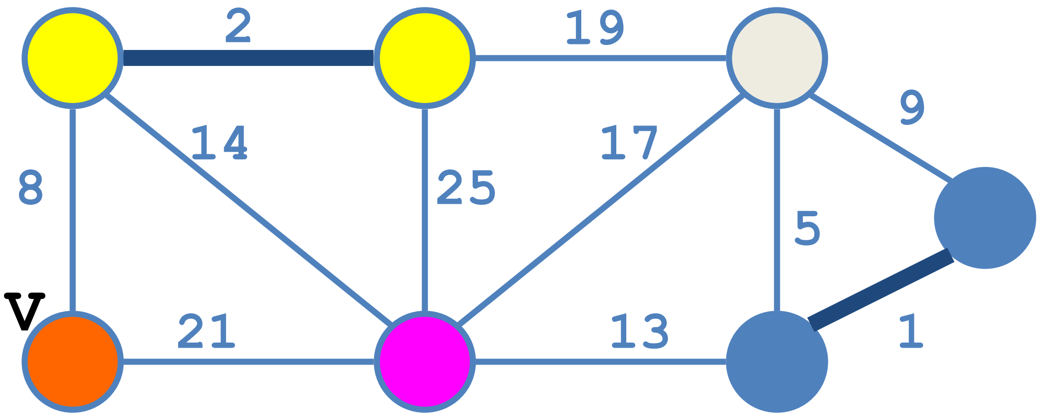
# Kruskal's Algorithm

```
Kruskal(G, w)
{
    A = ∅;
    for each v ∈ G.V
        Make-Set(v);
    sort G.E by non-decreasing order by weight w
    for each (u,v) ∈ G.E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            A = A U {{u,v}};
            Union(u, v);
}
```



# Kruskal's Algorithm

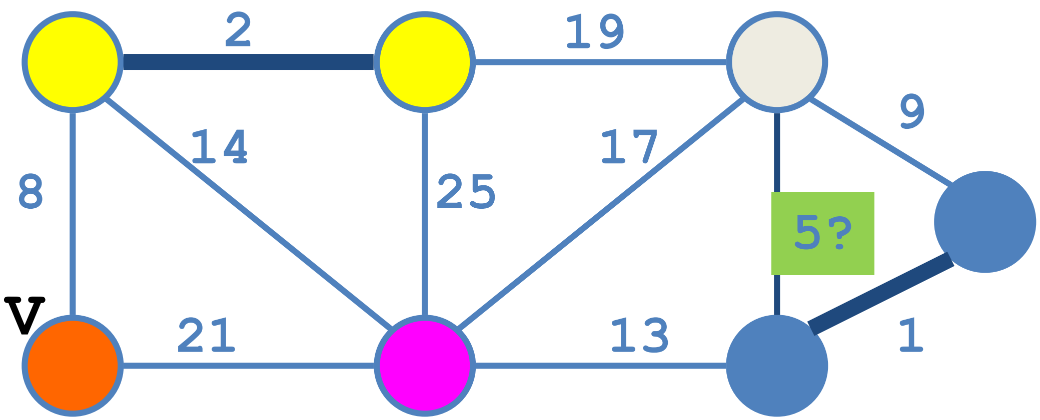
***Run the algorithm:***




```
Kruskal(G, w)
{
    A = ∅;
    for each v ∈ G.V
        Make-Set(v);
    sort G.E by non-decreasing order by weight w
    for each (u,v) ∈ G.E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            A = A U {{u,v}};
            Union(u, v);
}
```

# Kruskal's Algorithm

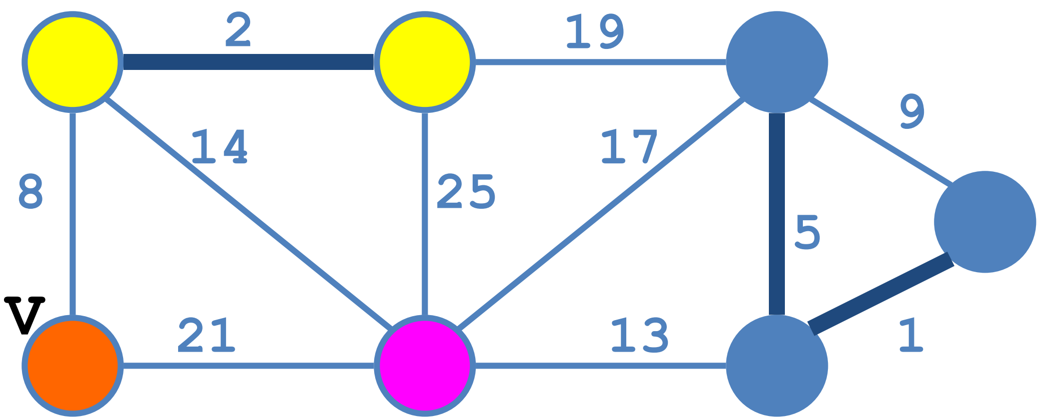
*Run the algorithm:*




```
Kruskal(G, w)
{
    A = ∅;
    for each v ∈ G.V
        Make-Set(v);
    sort G.E by non-decreasing order by weight w
    for each (u,v) ∈ G.E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            A = A U {{u,v}};
            Union(u, v);
}
```

# Kruskal's Algorithm

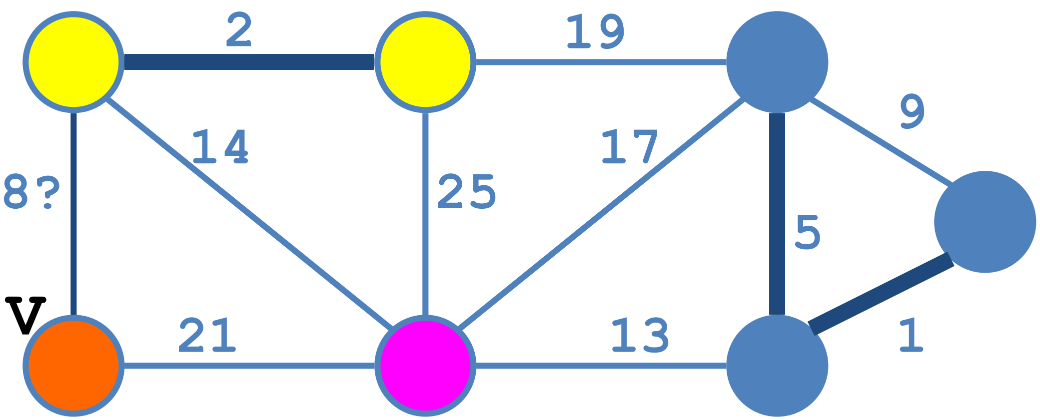
*Run the algorithm:*




```
Kruskal(G, w)
{
    A = ∅;
    for each v ∈ G.V
        Make-Set(v);
    sort G.E by non-decreasing order by weight w
    for each (u,v) ∈ G.E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            A = A U {{u,v}};
            Union(u, v);
}
```

# Kruskal's Algorithm

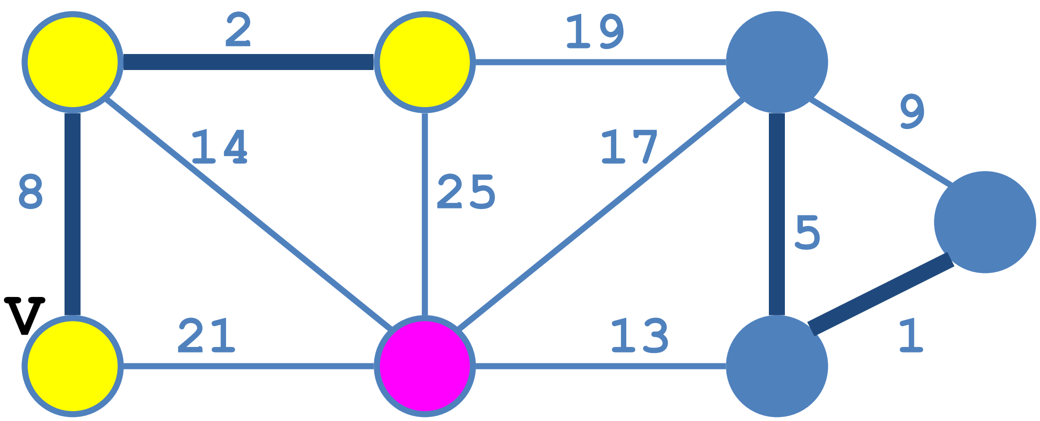
***Run the algorithm:***




```
Kruskal(G, w)
{
    A = ∅;
    for each v ∈ G.V
        Make-Set(v);
    sort G.E by non-decreasing order by weight w
    for each (u,v) ∈ G.E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            A = A U {{u,v}};
            Union(u, v);
}
```

# Kruskal's Algorithm

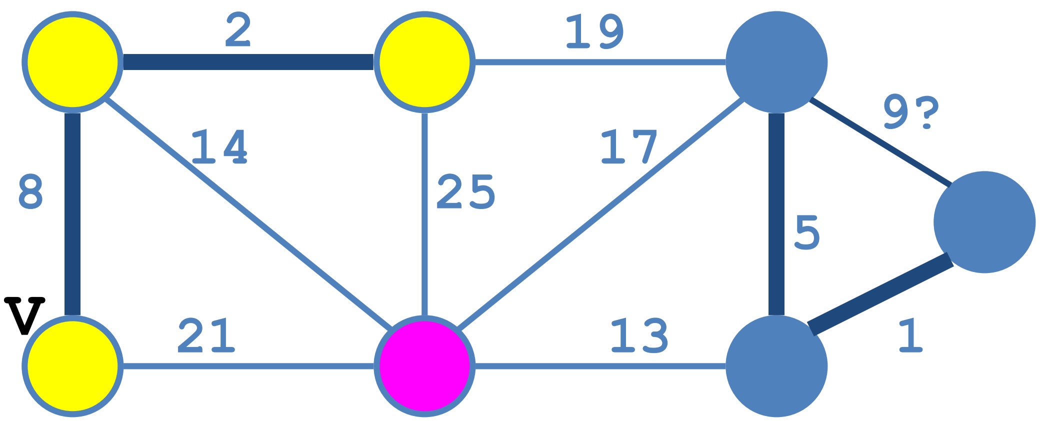
*Run the algorithm:*




```
Kruskal(G, w)
{
    A = ∅;
    for each v ∈ G.V
        Make-Set(v);
    sort G.E by non-decreasing order by weight w
    for each (u,v) ∈ G.E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            A = A U {{u,v}};
            Union(u, v);
}
```

# Kruskal's Algorithm

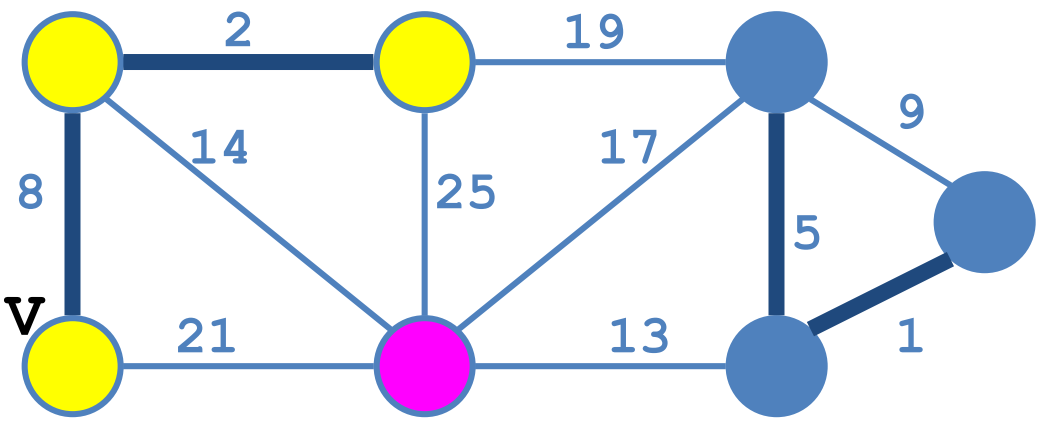
*Run the algorithm:*




```
Kruskal(G, w)
{
    A = ∅;
    for each v ∈ G.V
        Make-Set(v);
    sort G.E by non-decreasing order by weight w
    for each (u,v) ∈ G.E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            A = A U {{u,v}};
            Union(u, v);
}
```

# Kruskal's Algorithm

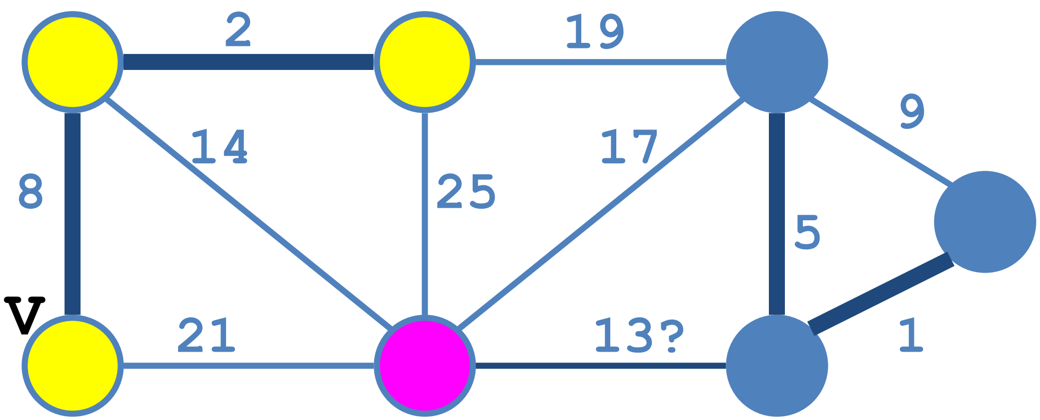
*Run the algorithm:*




```
Kruskal(G, w)
{
    A = ∅;
    for each v ∈ G.V
        Make-Set(v);
    sort G.E by non-decreasing order by weight w
    for each (u,v) ∈ G.E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            A = A U {{u,v}};
            Union(u, v);
}
```

# Kruskal's Algorithm

*Run the algorithm:*

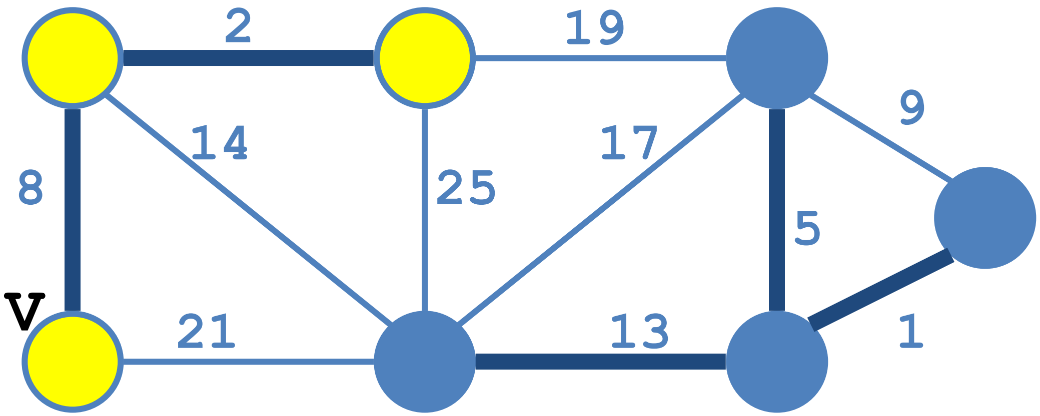



```
Kruskal(G, w)
{
    A = ∅;
    for each v ∈ G.V
        Make-Set(v);
    sort G.E by non-decreasing order by weight w
    for each (u,v) ∈ G.E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            A = A U {{u,v}};
            Union(u, v);
}
```

# Kruskal's Algorithm

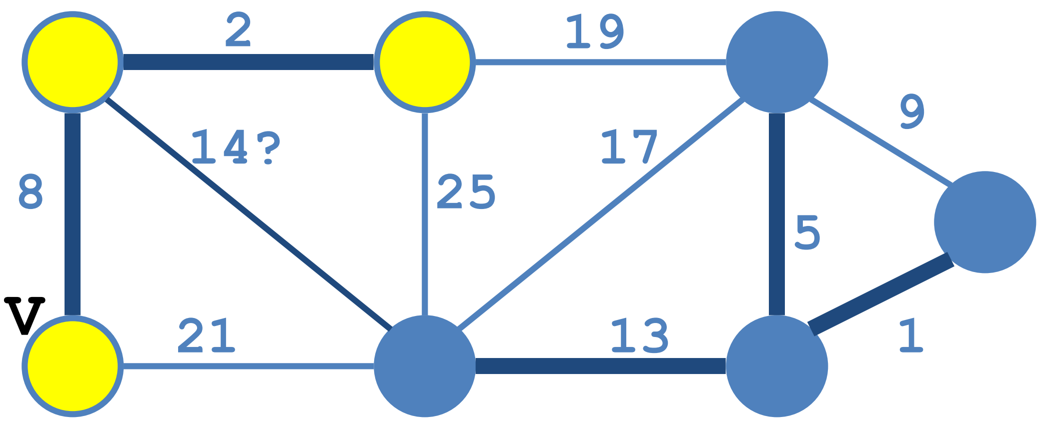
```
Kruskal(G, w)
{
    A = ∅;
    for each v ∈ G.V
        Make-Set(v);
    sort G.E by non-decreasing order by weight w
    for each (u,v) ∈ G.E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            A = A U {{u,v}};
            Union(u, v);
}
```

*Run the algorithm:*



# Kruskal's Algorithm

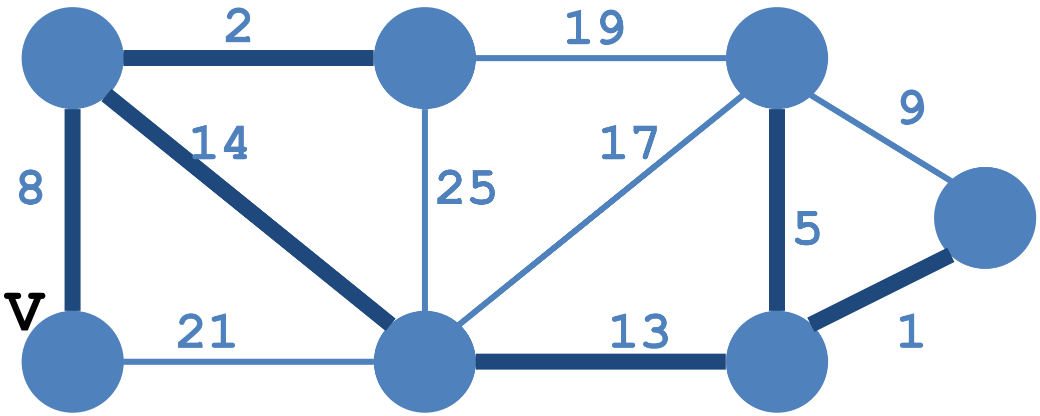
*Run the algorithm:*




```
Kruskal(G, w)
{
    A = ∅;
    for each v ∈ G.V
        Make-Set(v);
    sort G.E by non-decreasing order by weight w
    for each (u,v) ∈ G.E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            A = A U {{u,v}};
            Union(u, v);
}
```

# Kruskal's Algorithm

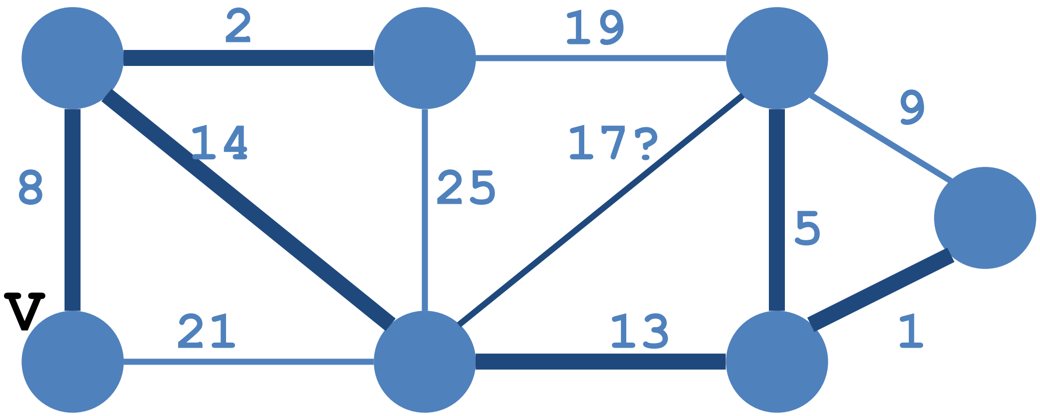
***Run the algorithm:***




```
Kruskal(G, w)
{
    A = ∅;
    for each v ∈ G.V
        Make-Set(v);
    sort G.E by non-decreasing order by weight w
  { for each (u,v) ∈ G.E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            A = A U {{u,v}};
            Union(u, v);
}
```

# Kruskal's Algorithm

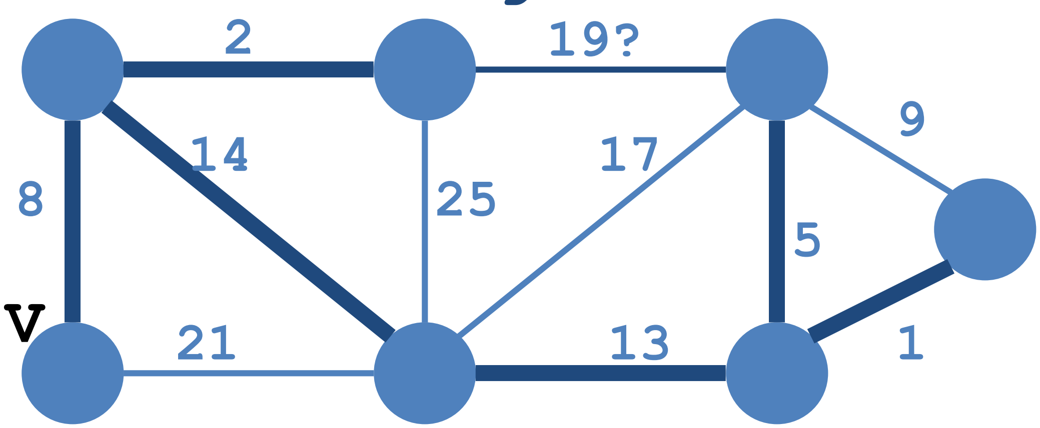
***Run the algorithm:***




```
Kruskal(G, w)
{
    A = ∅;
    for each v ∈ G.V
        Make-Set(v);
    sort G.E by non-decreasing order by weight w
    for each (u,v) ∈ G.E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            A = A U {{u,v}};
            Union(u, v);
}
```

# Kruskal's Algorithm

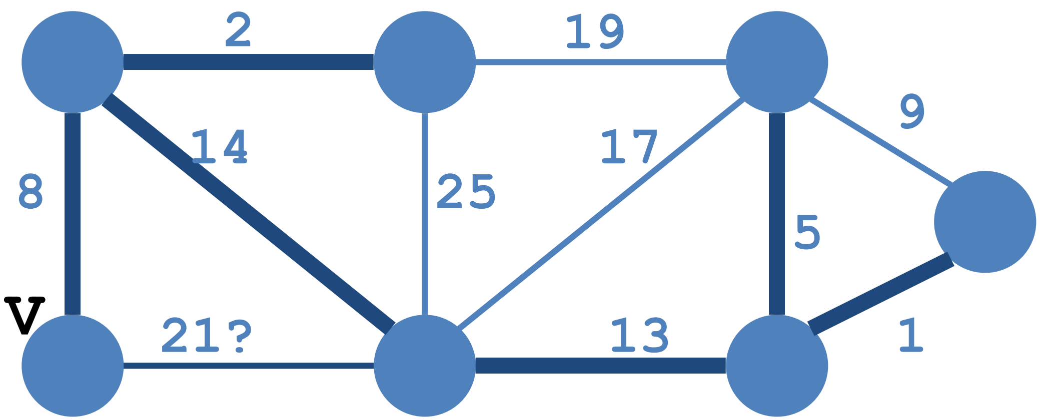
***Run the algorithm:***




```
Kruskal(G, w)
{
    A = ∅;
    for each v ∈ G.V
        Make-Set(v);
    sort G.E by non-decreasing order by weight w
    for each (u,v) ∈ G.E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            A = A ∪ {{u,v}};
            Union(u, v);
}
```

# Kruskal's Algorithm

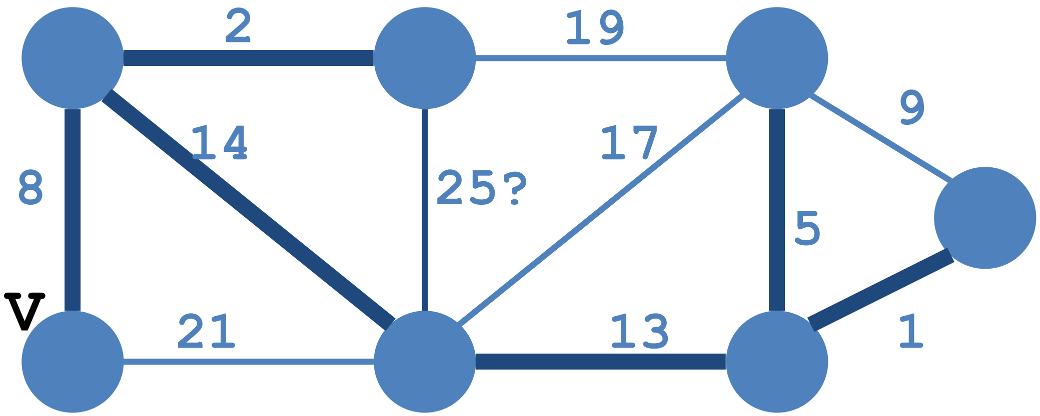
***Run the algorithm:***




```
Kruskal(G, w)
{
    A = ∅;
    for each v ∈ G.V
        Make-Set(v);
    sort G.E by non-decreasing order by weight w
    for each (u,v) ∈ G.E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            A = A U {{u,v}};
            Union(u, v);
}
```

# Kruskal's Algorithm

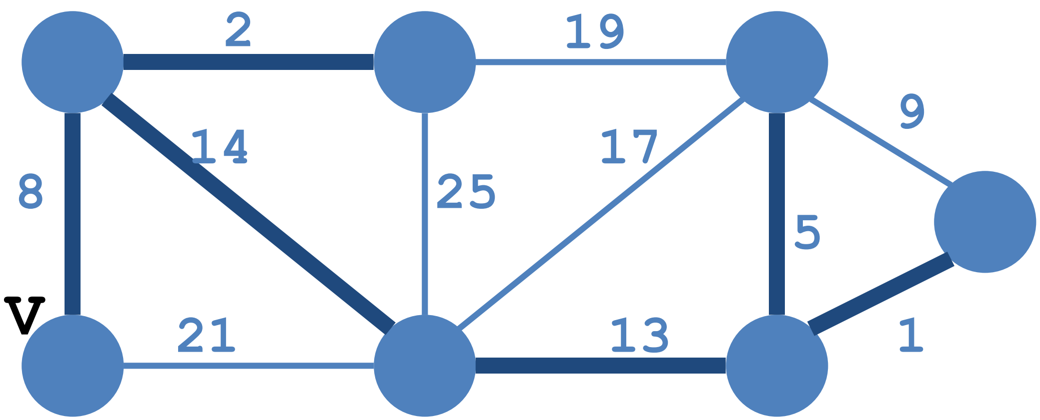
***Run the algorithm:***




```
Kruskal(G, w)
{
    A = ∅;
    for each v ∈ G.V
        Make-Set(v);
    sort G.E by non-decreasing order by weight w
    for each (u,v) ∈ G.E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            A = A U {{u,v}};
            Union(u, v);
}
```

# Kruskal's Algorithm: Done

***Run the algorithm:***




```
Kruskal(G, w)
{
    A = ∅;
    for each v ∈ G.V
        Make-Set(v);
    sort G.E by non-decreasing order by weight w
    for each (u,v) ∈ G.E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            A = A U {{u,v}};
            Union(u, v);
}
```

# Correctness Of Kruskal's Algorithm

- Sketch of a proof: this algorithm produces an MST of *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 another lower weight edge
  - But algorithm chooses lowest weight edge at each step. Contradiction

# Kruskal's Algorithm

```
Kruskal(G, w)
{
    A = ∅;
    for each v ∈ G.V
        Make-Set(v);
    sort G.E by non-decreasing order by weight w
    for each (u,v) ∈ G.E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            A = A U {{u,v}};
            Union(u, v);
}
```

***What will affect the running time?***

| | |
|---|---|
| **Initialize A** | **O(1)** |
| **1st FOR loop** | **\|V\| MakeSet() calls** |
| **Sort** | **O(E lgE)** |
| **FINDSET()/Union()** | **O(E) calls** |

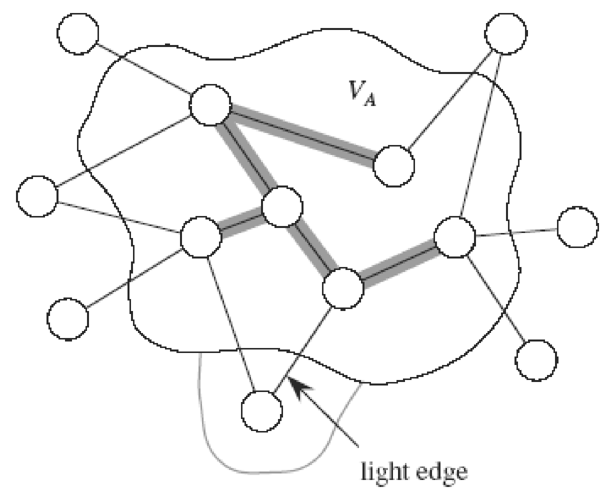
# Kruskal's Algorithm: Running Time

- Initialize A: O(1)
- First for loop: |V| MAKE-SETs
- Sort E: O(E lg E)
- Second for loop: O(E) FIND-SETs and UNIONs
- **O(V) +O (E α(V)) + O(E lg E)**
  - Since G is connected, $|E| \geq |V| - 1 \Rightarrow O(E\ \alpha(V)) + O(E \lg E)$
  - $\alpha(|V|) = O(\lg V) = O(\lg E)$
  - Therefore, the total time is O(E lg E)
  - $|E| \leq |V|^2 \Rightarrow \lg |E| = O(2 \lg V) = O(\lg V)$
  - Therefore, O(E lg V) time

# Prim's algorithm

- Build a tree *A*
  - Starts from an arbitrary "root" *r*.
  - At each step, find a <u>light edge</u> crossing the cut $(V_A, V - V_A)$, where $V_A$ = vertices that *A* is incident on.
  - Add this light edge to *A*.
- GREEDY CHOICE*:*

  *add min weight to A*




[Edges of *A* are shaded.]

# How to find the light edge quickly?

- Use a priority queue $Q$
  - Each object is a vertex in $V - V_A$
  - Key of $v$ is the minimum weight of any edge $(u, v)$, where $u \in V_A$
  - the vertex returned by EXTRACT-MIN is $v$
    - such that there exists $u \in V_A$, and edge $(u, v)$ is a light edge crossing $(V_A, V - V_A)$
- Key of $v$ is $\infty$, if $v$ is not adjacent to any vertices in $V_A$

# How to find the light edge quickly?

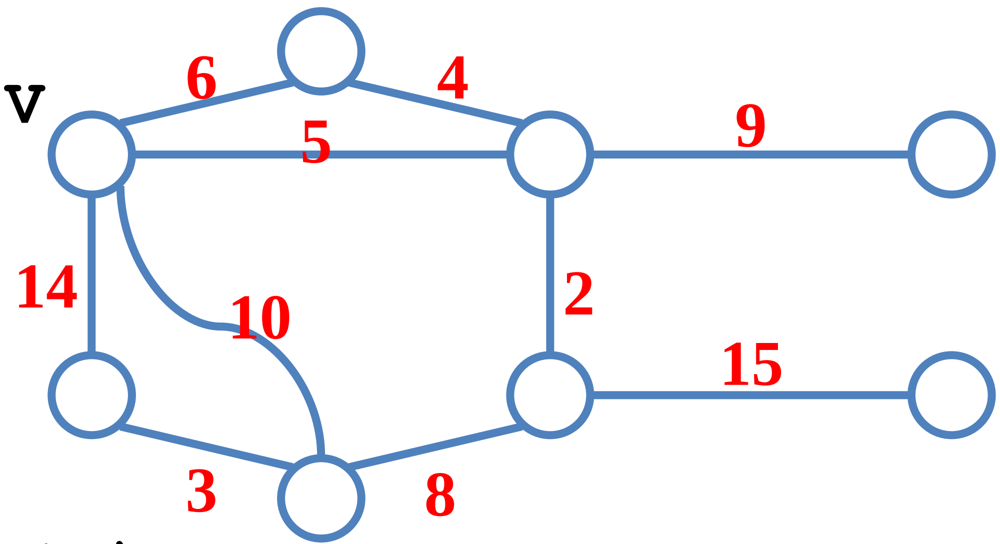
- The edges of $A$ form a rooted tree with root $r$
  - $r$ is given as an input to the algorithm, but it can be any vertex
  - Each vertex knows its parent in the tree by the attribute v.$\pi$ = parent of $v$
  - $\pi[v]$ = NIL, if $v = r$ or $v$ has no parent.
  - As the algorithm progresses, $A = \{(v, v.\pi) : v \in V - \{r\} - Q\}$

# Prim's Algorithm

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

# Prim's Algorithm

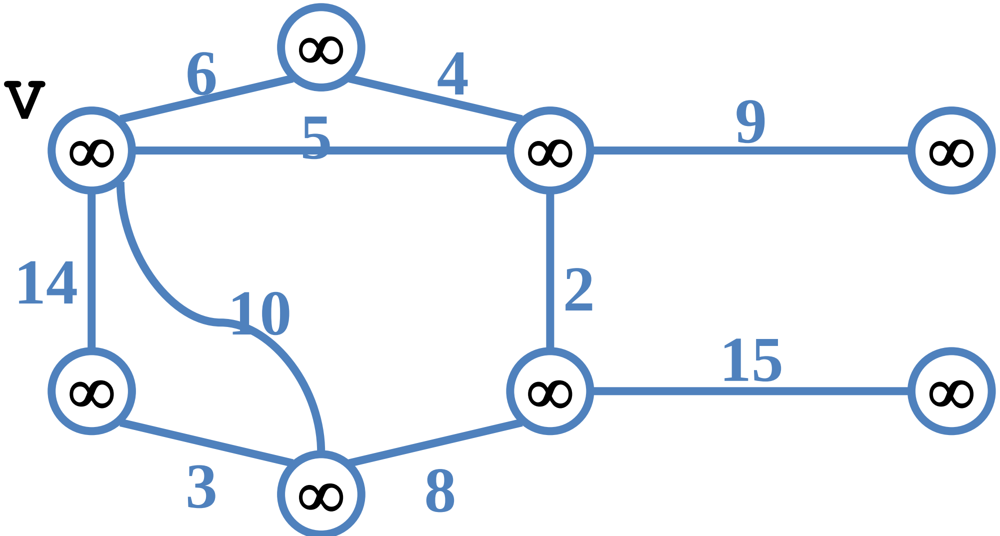
```
MST-Prim(G, w, r)
    for each u ∈ G.V
        u.key = ∞
        u.π = NIL
    r.key = 0
    Q = G.V
    while (Q not empty)
        u = ExtractMin(Q)
        for each v ∈ G.Adj[u]
            if (v ∈ Q and w(u,v) < v.key)
                v.π = u
                v.key = w(u,v)
```




Run on example graph

# Prim's Algorithm

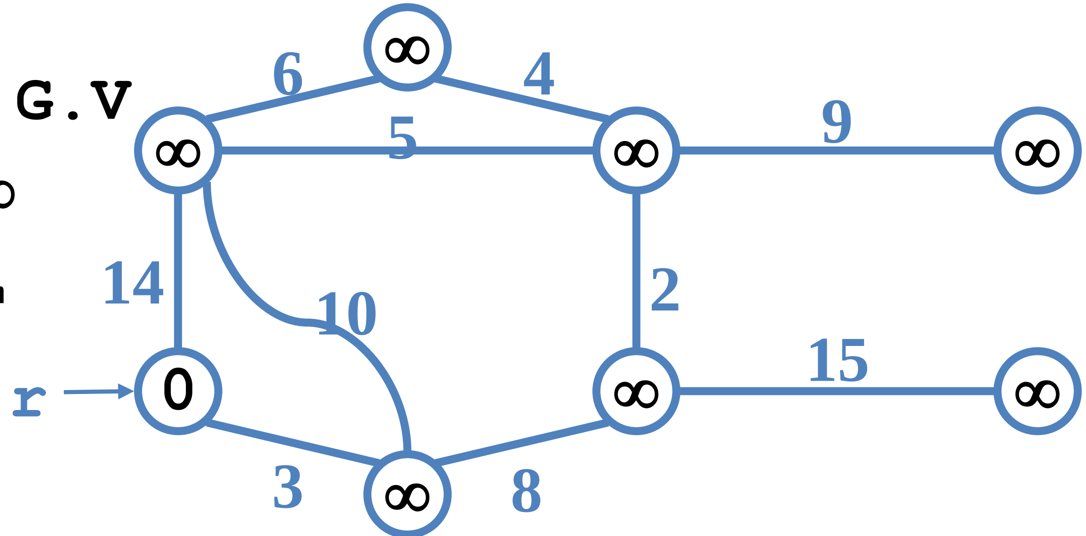
```
MST-Prim(G, w, r)
    for each u ∈ G.V
        u.key = ∞
        u.π = NIL
    r.key = 0
    Q = G.V
    while (Q not empty)
        u = ExtractMin(Q)
        for each v ∈ G.Adj[u]
            if (v ∈ Q and w(u,v) < v.key)
                v.π = u
                v.key = w(u,v)
```




Run on example graph

# Prim's Algorithm

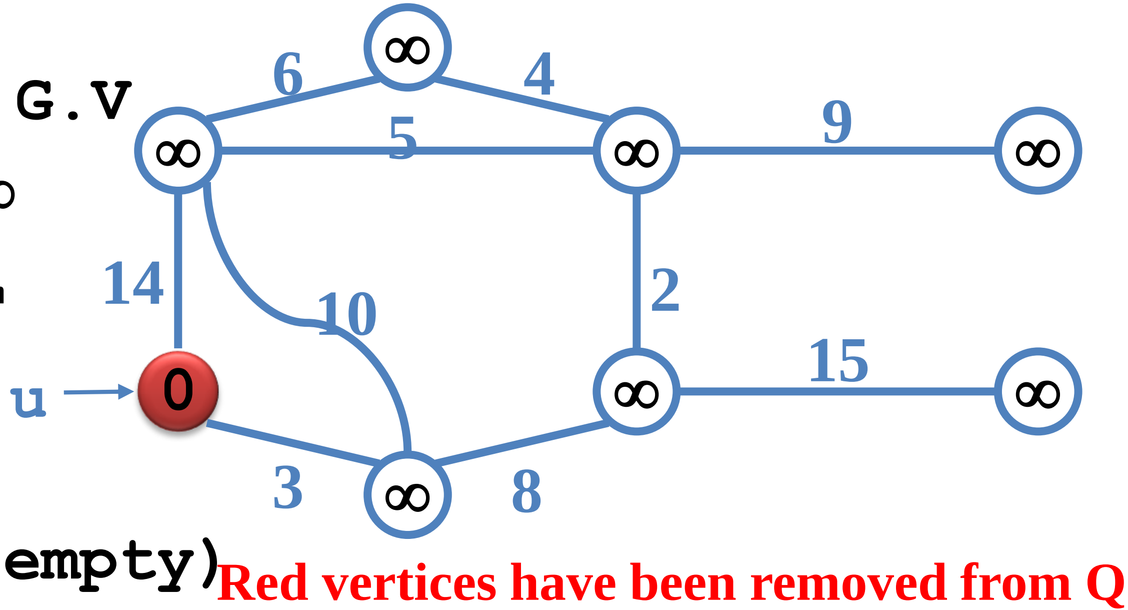
```
MST-Prim(G, w, r)
    for each u ∈ G.V
        u.key = ∞
        u.π = NIL
    r.key = 0
    Q = G.V
    while (Q not empty)
        u = ExtractMin(Q)
        for each v ∈ G.Adj[u]
            if (v ∈ Q and w(u,v) < v.key)
                v.π = u
                v.key = w(u,v)
```




Pick a start vertex r

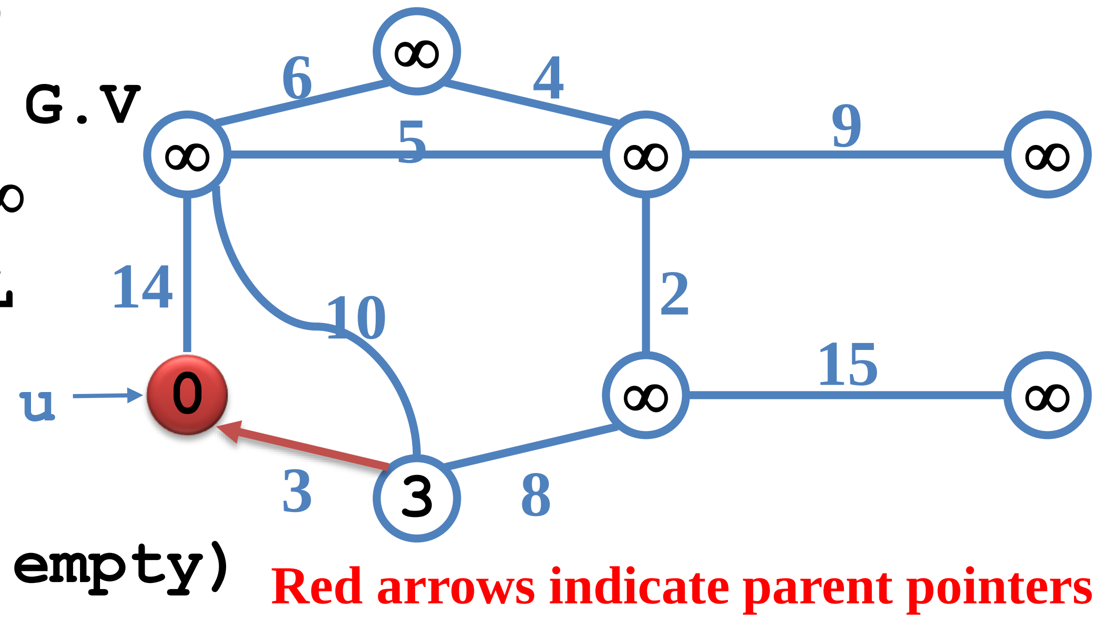
# Prim's Algorithm

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



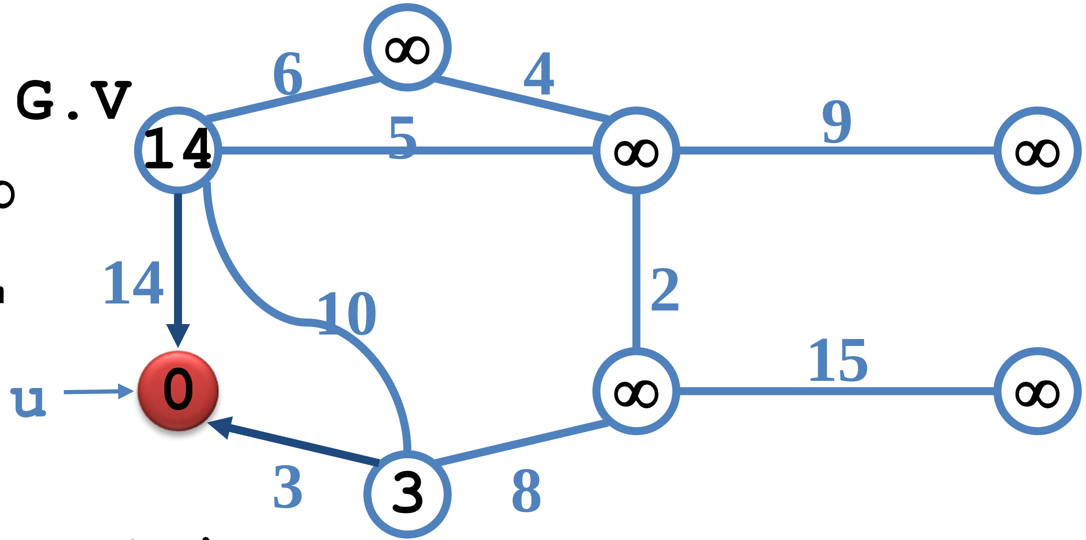
# Prim's Algorithm

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



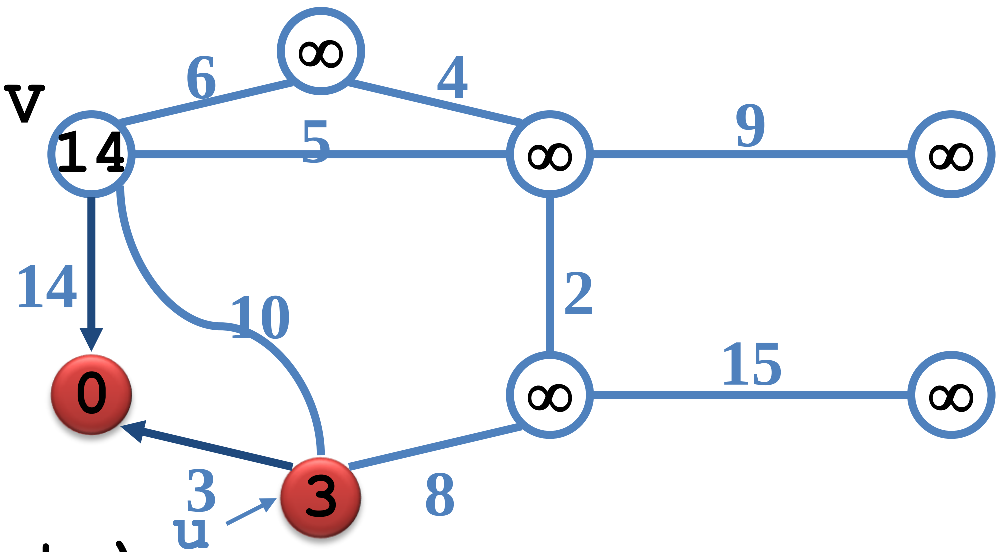
# Prim's Algorithm

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



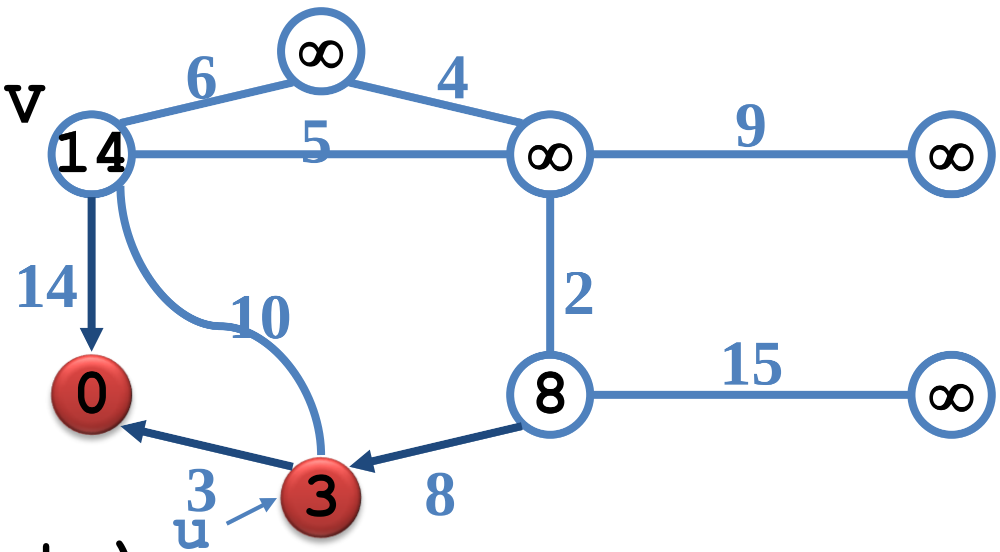
# Prim's Algorithm

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



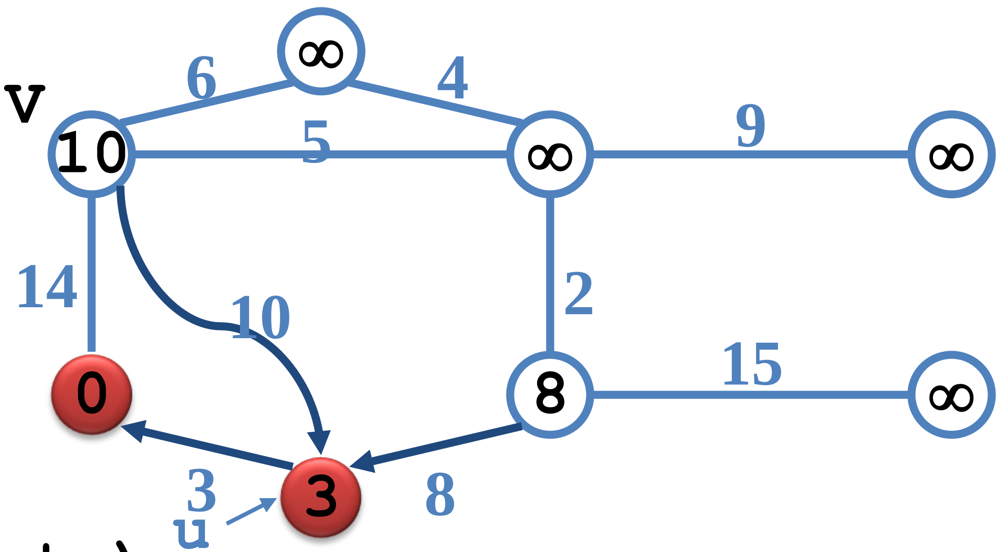
# Prim’s Algorithm

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



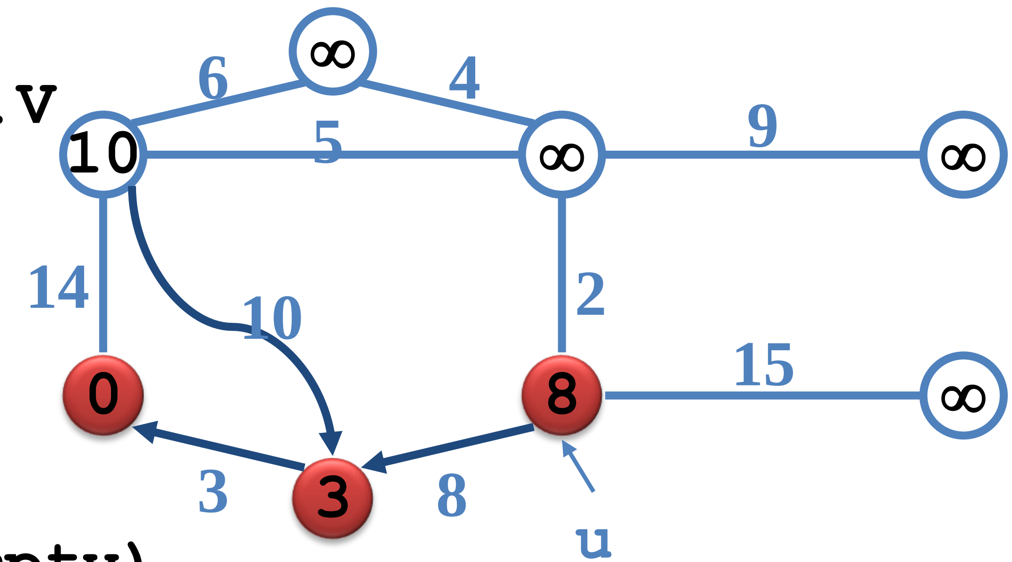
# Prim's Algorithm

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



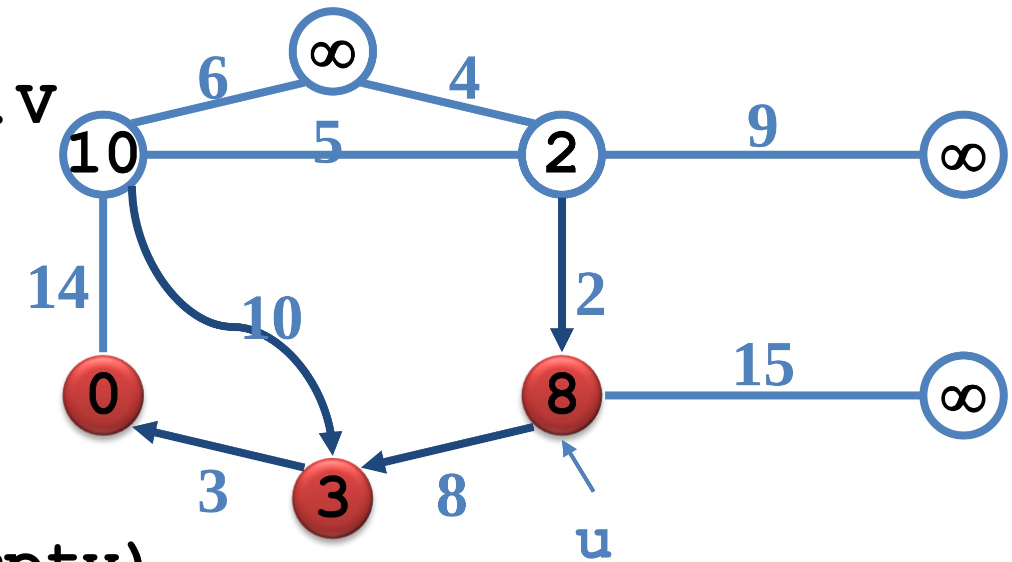
# Prim's Algorithm

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



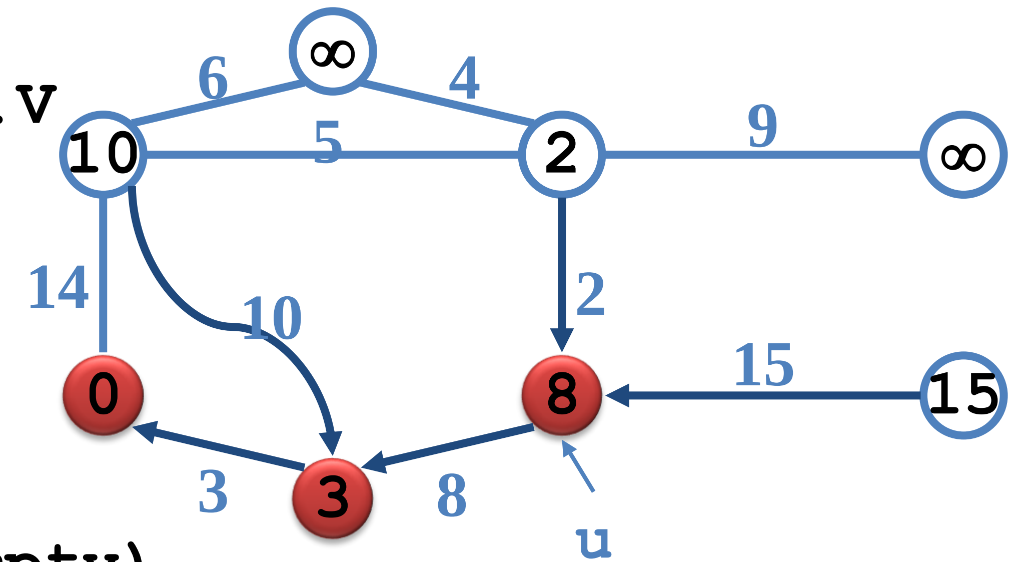
# Prim's Algorithm

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



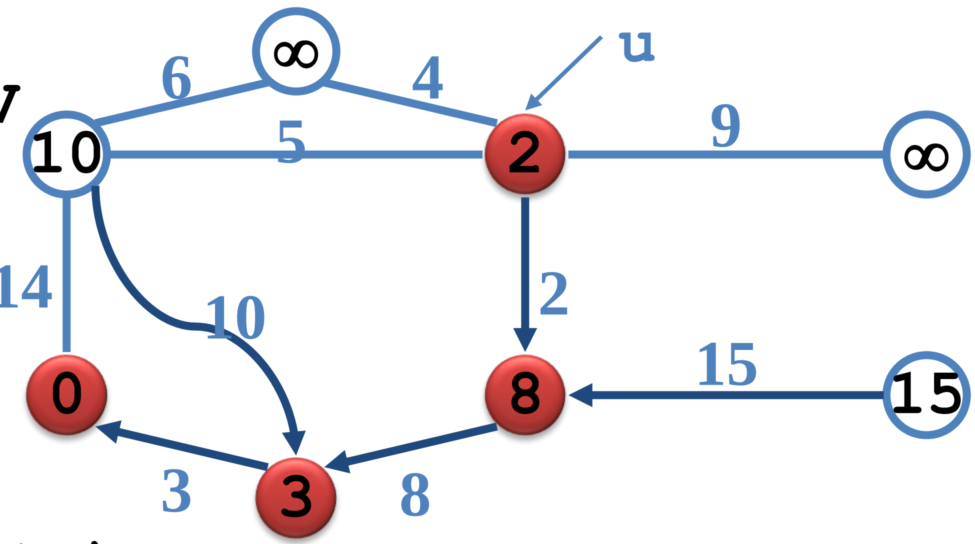
# Prim's Algorithm

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



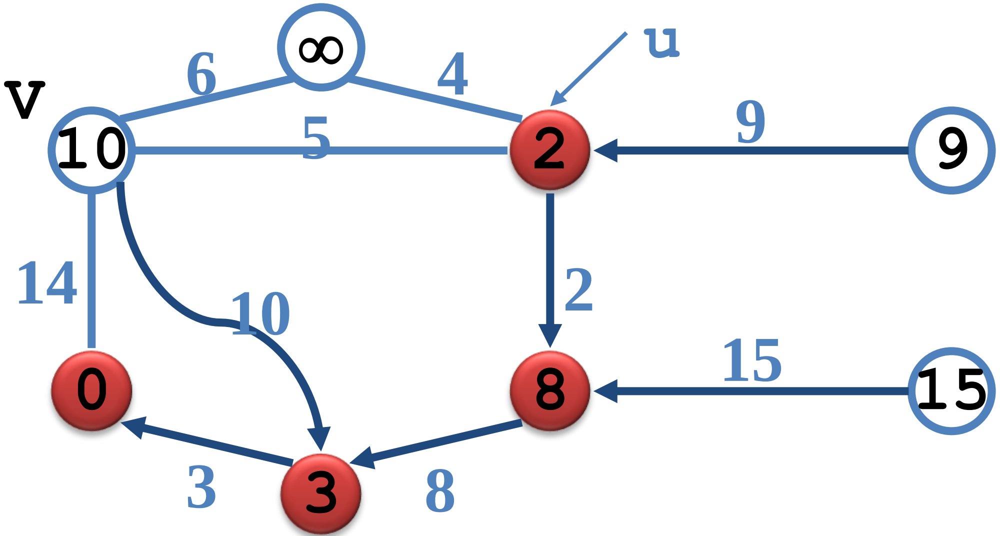
# Prim's Algorithm

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



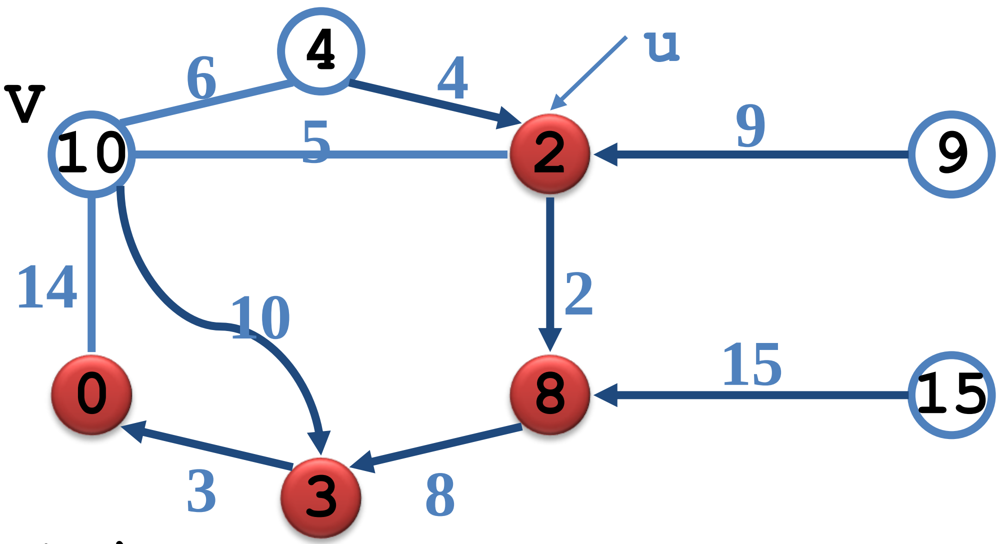
# Prim's Algorithm

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



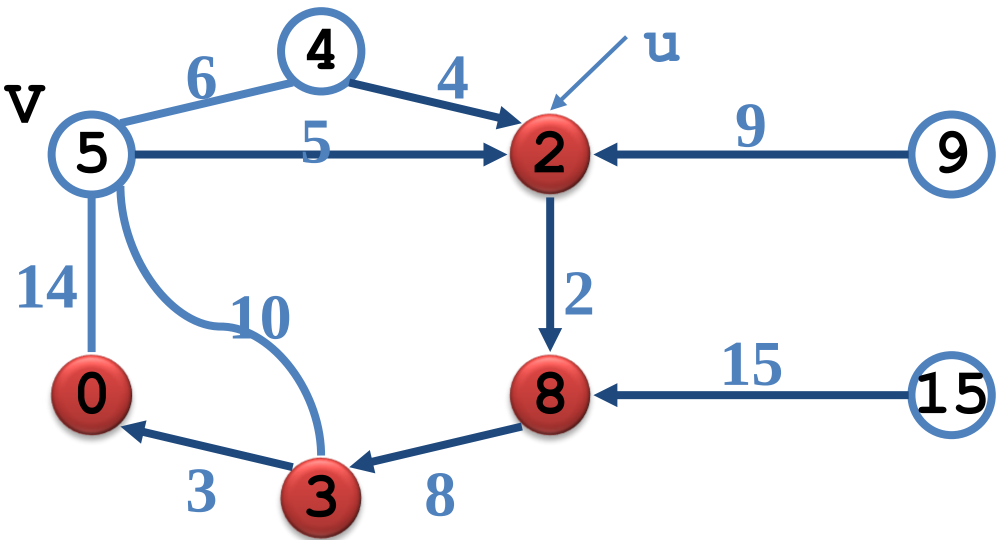
# Prim's Algorithm

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



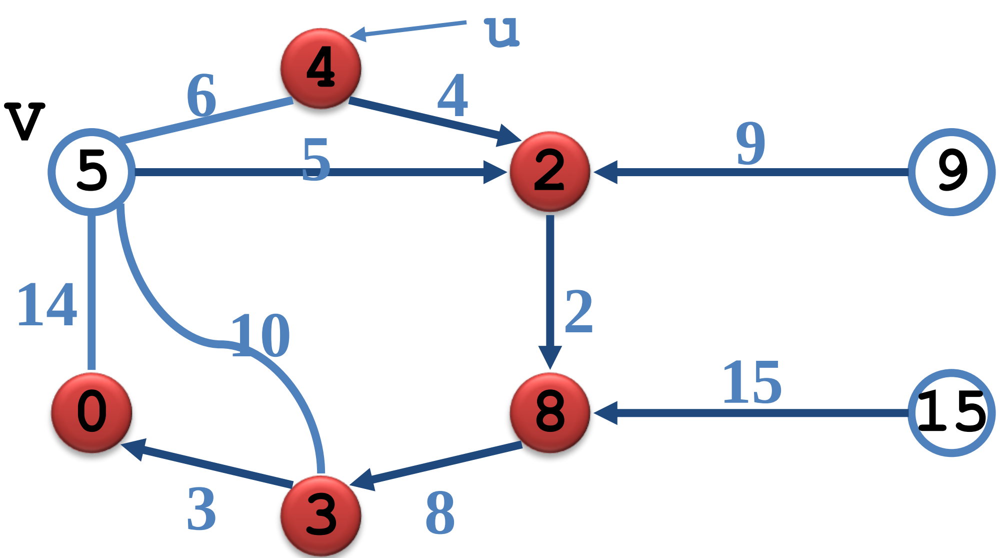
# Prim's Algorithm

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



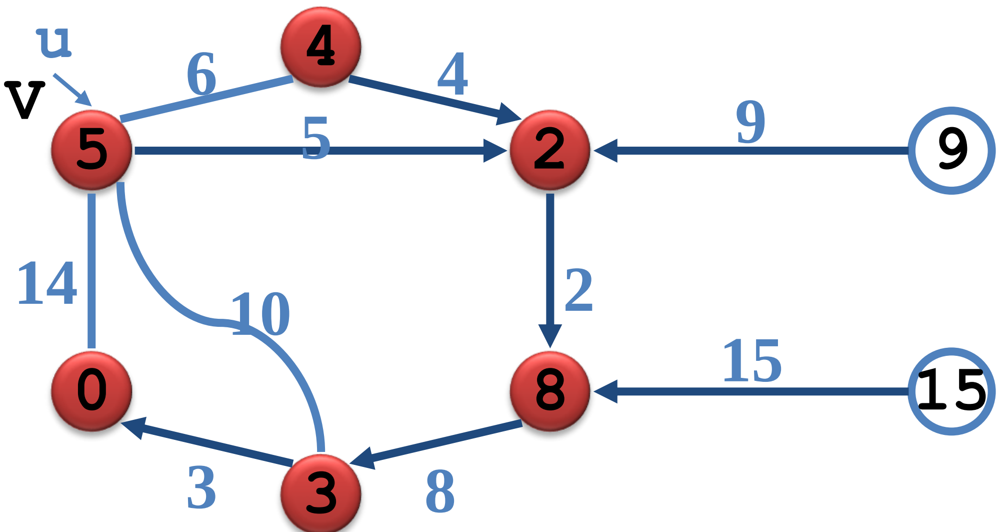
# Prim's Algorithm

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



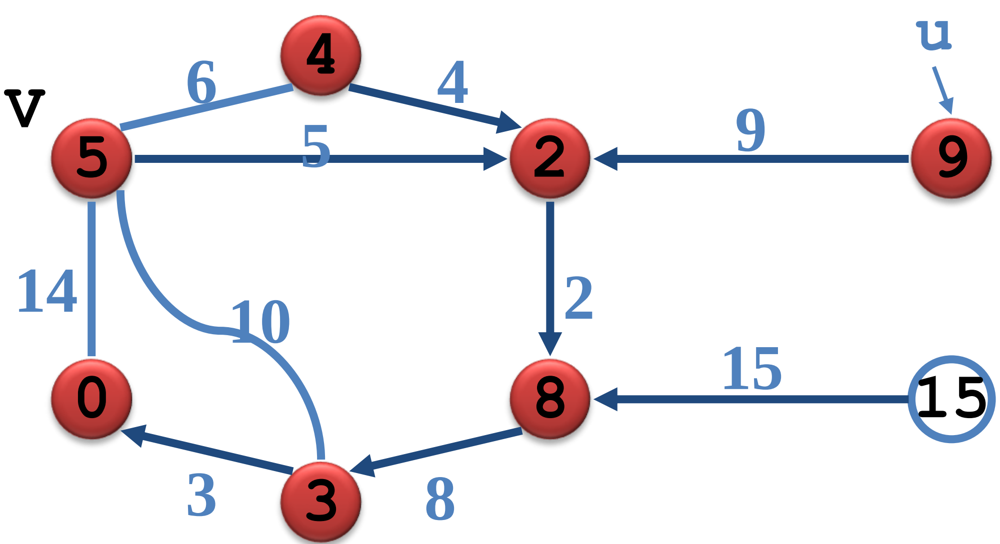
# Prim's Algorithm

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



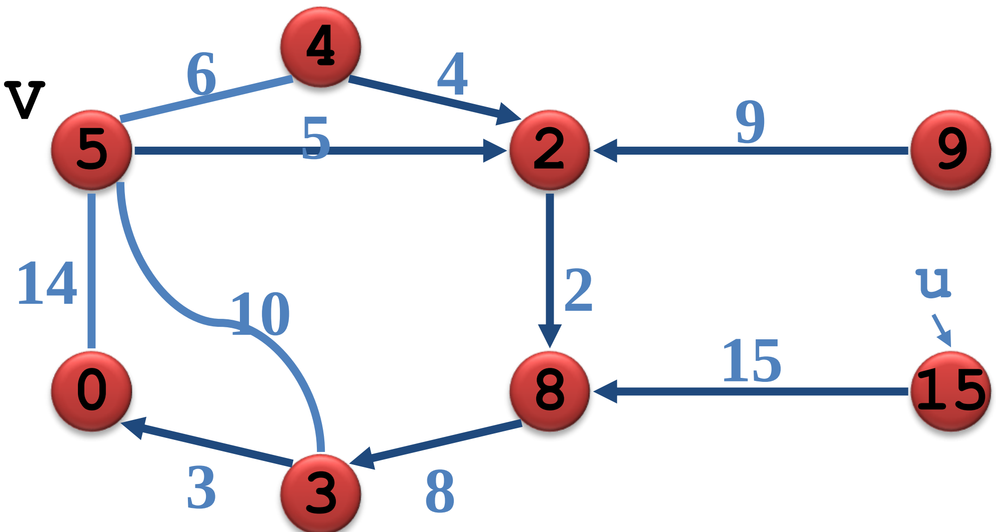
# Prim's Algorithm

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

**What is the hidden cost in this code?**

# Review: Prim's Algorithm

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

# Prim's Algorithm: running time

- We can use the BUILD-MIN-HEAP procedure to perform the initialization in lines 1–5 in $O(V)$ time
- EXTRACT-MIN operation is called $|V|$ times, and each call takes $O(\lg V)$ time, the total time for all calls to EXTRACT-MIN is $O(V \lg V)$

# Running time (cont’d)

- The for loop in lines 8–11 is executed O(E) times altogether, since the sum of the lengths of all adjacency lists is 2 |E|.
  - Lines 9 -10 take constant time
  - line 11 involves an implicit DECREASE-KEY operation on the min-heap, which takes $O(\lg V)$ time
- Thus, the total time for Prim's algorithm is $O(V) + O(V \lg V) + O(E \lg V) = O(E \lg V)$
  - The same as Kruskal's algorithm

# Summary

- We learned
  - Generic MST
  - Kruskal’s and Prim’s algorithm
- Common mistakes: Don’t mix Kruskal’s algorithm with Prim’s algorithm