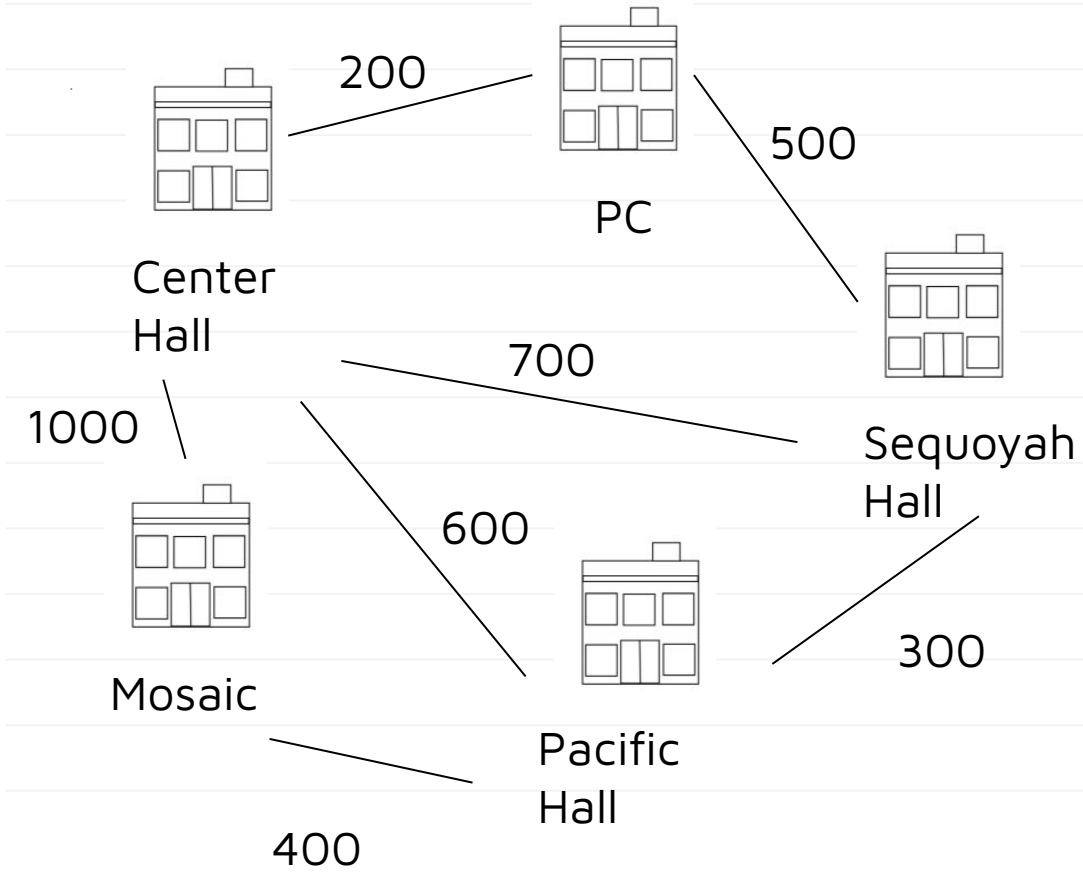
# DSC 40B

## Lecture 26 : Minimum Spanning Trees. Prim's algorithm

# Minimum Spanning Trees

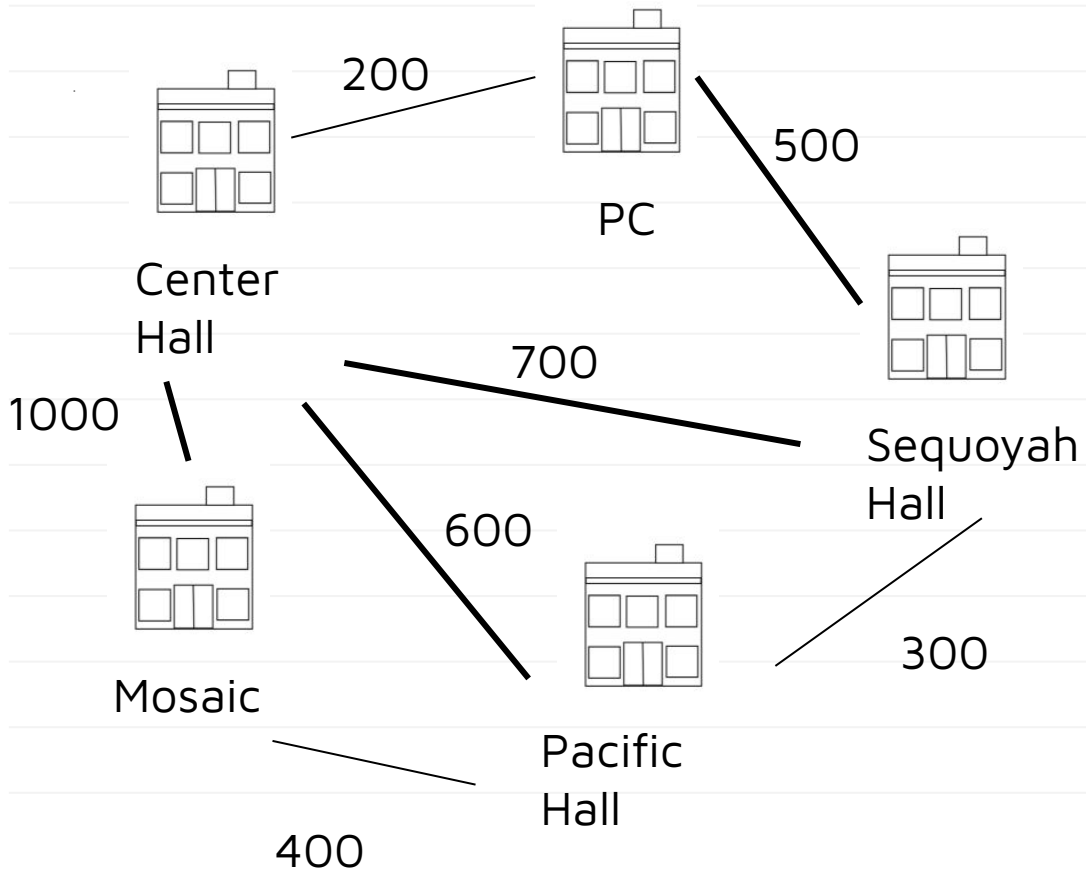
# Dream World




Choose a set of roads so that:

- Every building must get coffee.
- No loops
- Total cost is minimized.

# Dream World



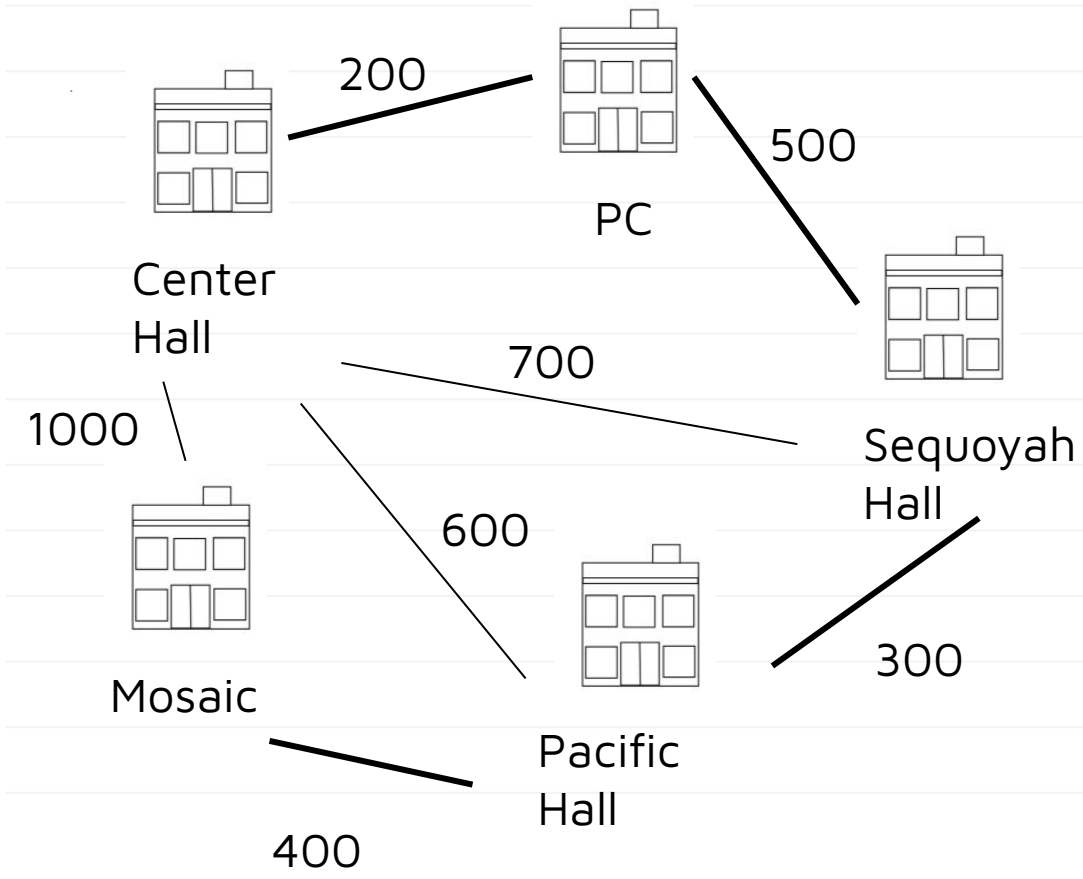

Choose a set of roads so that:

- Every building must get coffee.
- No loops
- Total cost is minimized.

One way:

- 1000 + 700 + 500 + 600=2800
- Not the shortest

# Dream World




Choose a set of roads so that:

- Every building must get coffee.
- No loops
- Total cost is minimized.

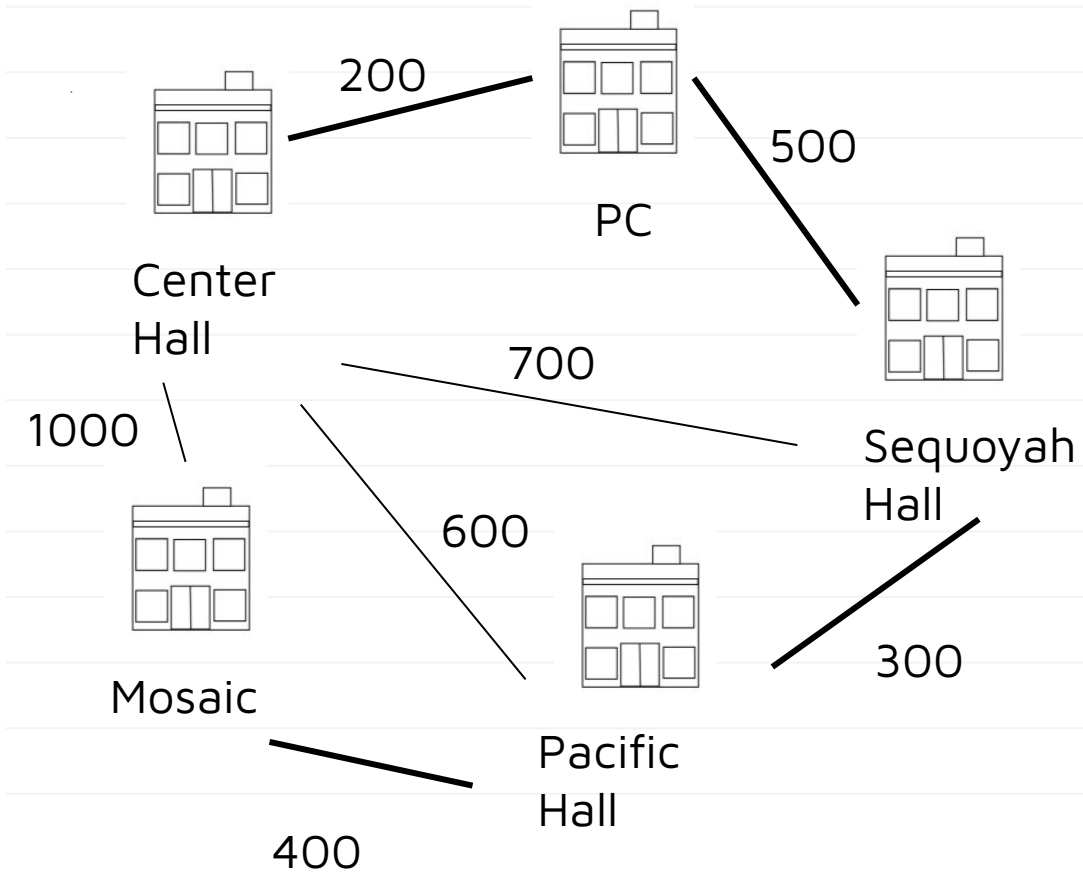
One way:

- 1000 + 700 + 500 + 600=2800
- **Not the shortest**

Better path:

- 200 + 300 + 400 + 500=1400

# Dream World




Choose a set of roads so that:

- Every building must get coffee.
- No loops
- Total cost is minimized.

One way:

- 1000 + 700 + 500 + 600=2800
- **Not the shortest**

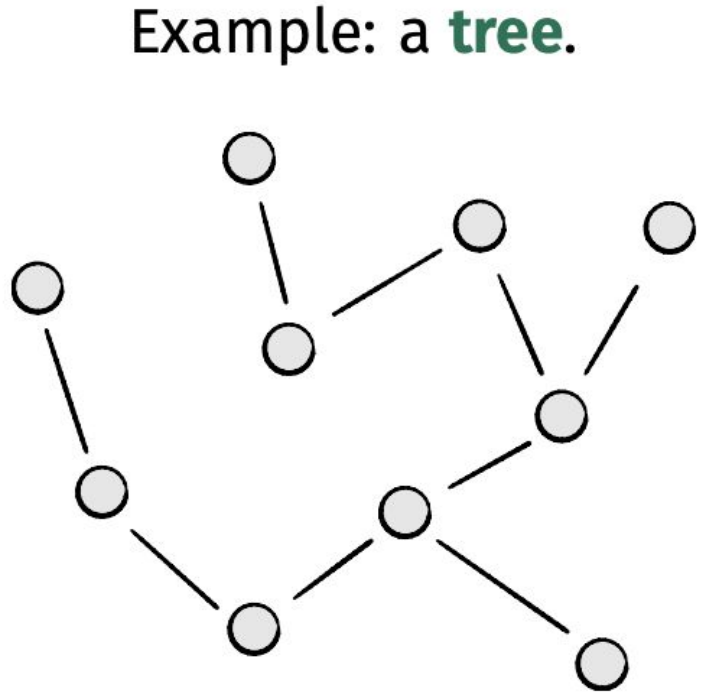
Better path:

- 200 + 300 + 400 + 500=1400

**Solution**: compute a **minimum spanning tree.**

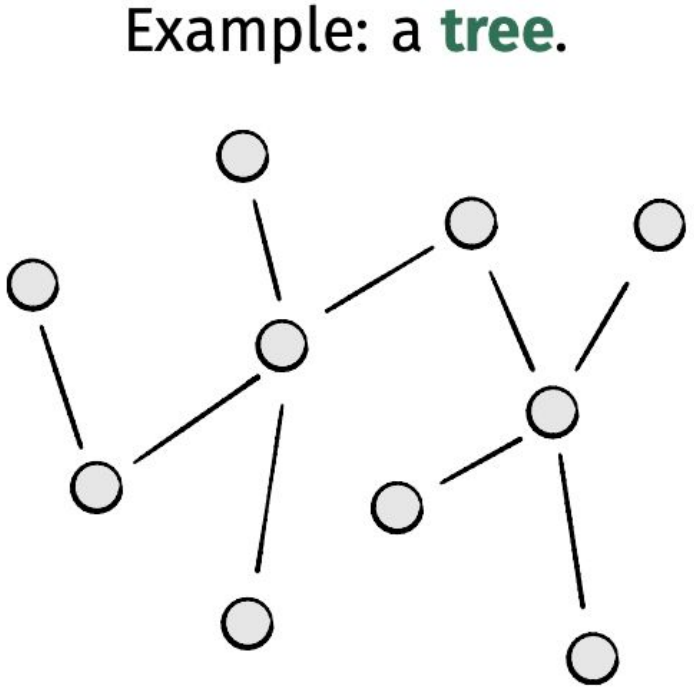
# Trees

- An undirected graph $T = (V, E)$ is a tree if
  - it is connect and
  - It is acyclic



# Trees

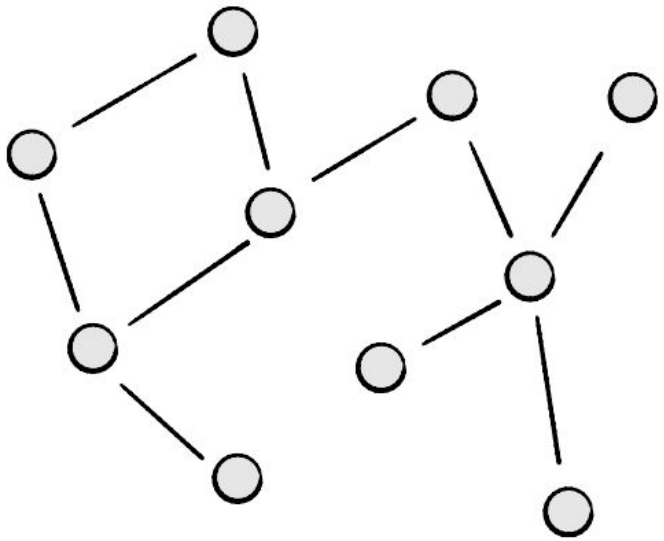
- An undirected graph $T = (V, E)$ is a tree if
  - it is connect and
  - It is acyclic



# Trees

- An undirected graph $T = (V, E)$ is a tree if
  - it is connect and
  - It is acyclic

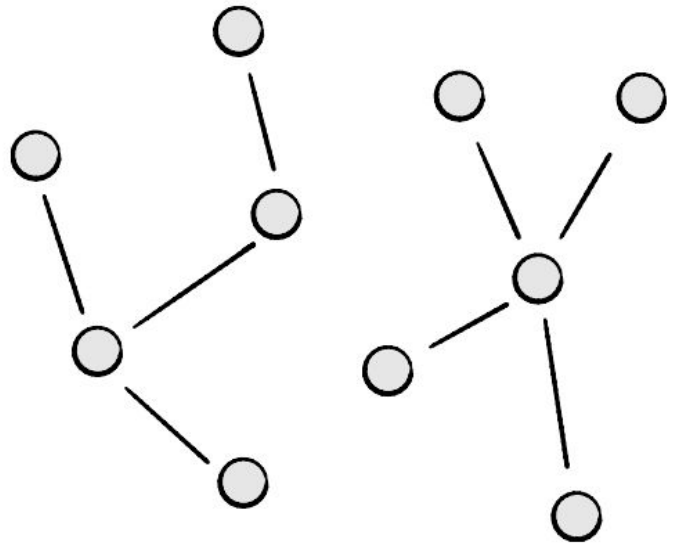
Example: **not** a tree.



# Trees

- An undirected graph $T = (V, E)$ is a tree if
  - it is connect and
  - It is acyclic

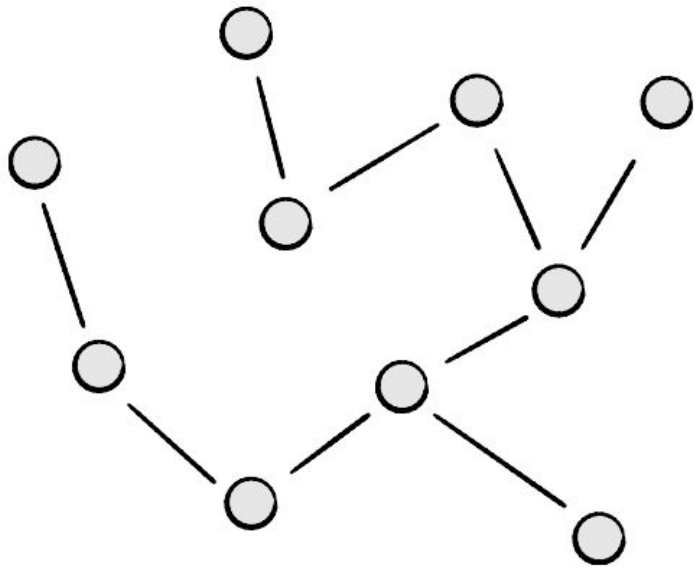
Example: **not** a tree.



# Trees: Equivalent Definition

- An undirected graph $T = (V, E)$ is a tree if
  - it is connected; and
  - $|E| = |V| - 1$.

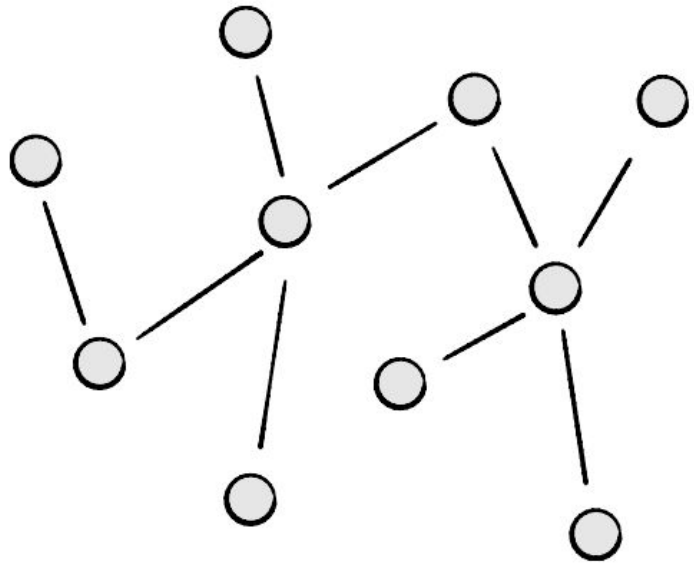
Example: a **tree**.



# Trees: Equivalent Definition

- An undirected graph $T = (V, E)$ is a tree if
  - it is connected; and
  - $|E| = |V| - 1$.

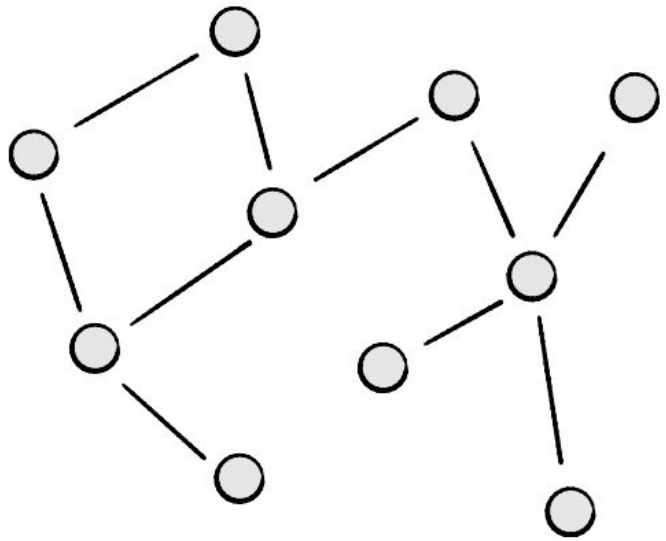
Example: a **tree**.



# Trees: Equivalent Definition

- An undirected graph $T = (V, E)$ is a tree if
  - it is connected; and
  - $|E| = |V| - 1$.

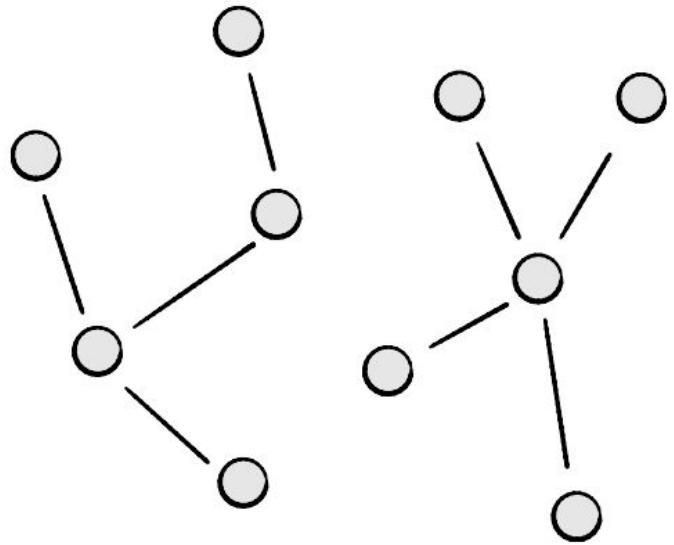
Example: **not** a tree.



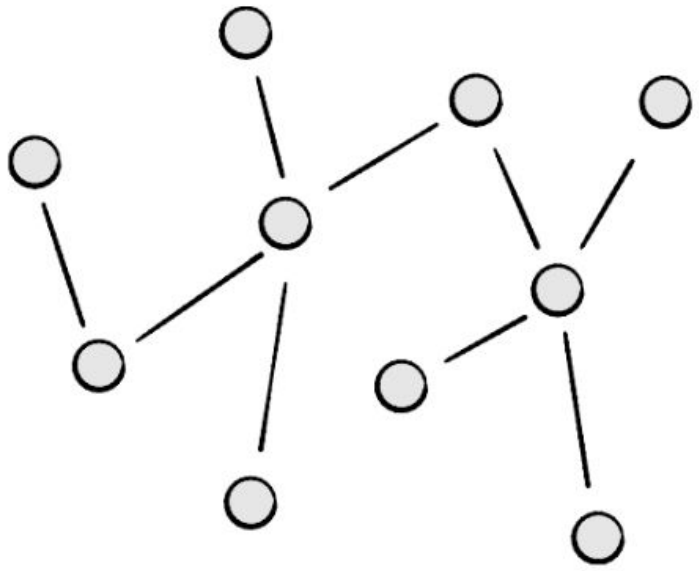
# Trees: Equivalent Definition

- An undirected graph $T = (V, E)$ is a tree if
  - it is connected; and
  - $|E| = |V| - 1$.

Example: **not** a tree.

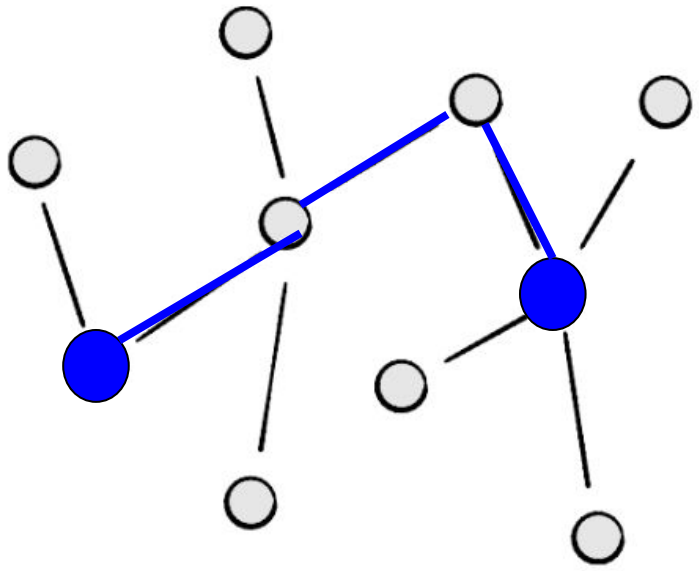


# Tree Properties



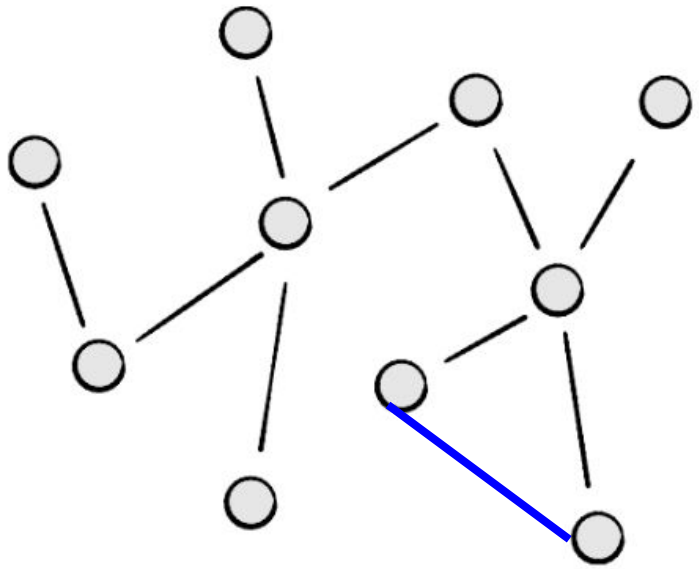
- There is a unique simple path between any two nodes in a tree.
- Adding a new edge to a tree creates a cycle (no longer a tree).
- Removing an edge from a tree disconnects it (no longer a tree).

# Tree Properties



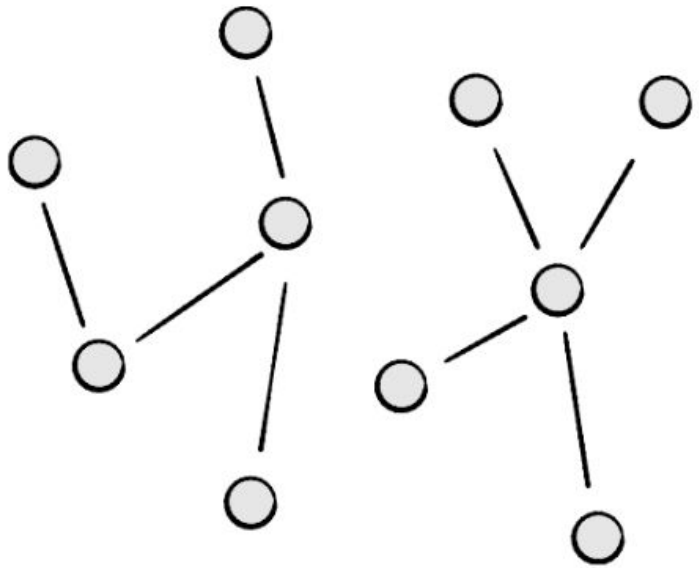
- **There is a unique simple path between any two nodes in a tree.**
- Adding a new edge to a tree creates a cycle (no longer a tree).
- Removing an edge from a tree disconnects it (no longer a tree).

# Tree Properties



- There is a unique simple path between any two nodes in a tree.
- **Adding a new edge to a tree creates a cycle (no longer a tree).**
- Removing an edge from a tree disconnects it (no longer a tree).

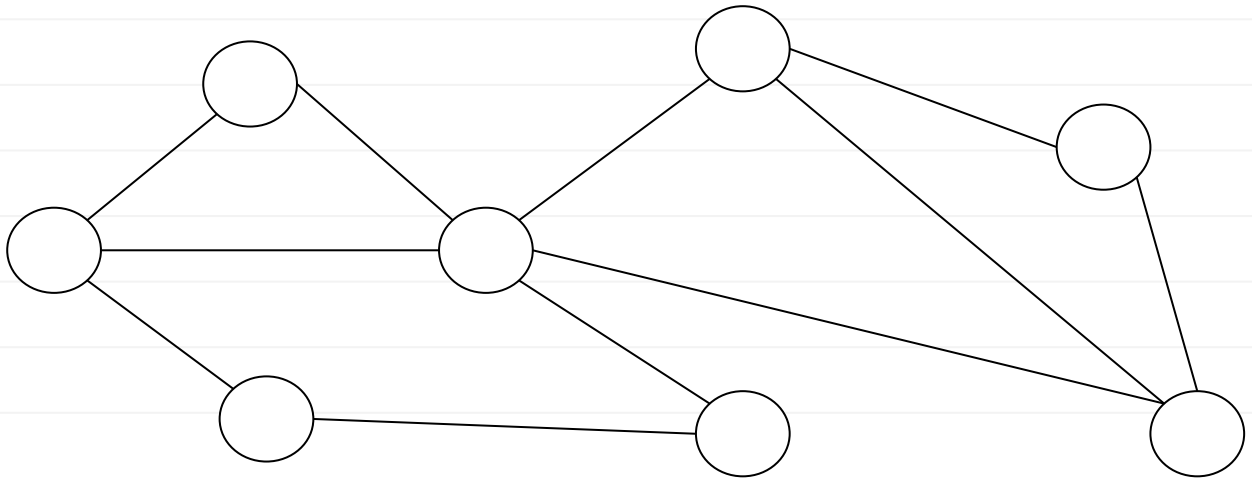
# Tree Properties



- There is a unique simple path between any two nodes in a tree.
- Adding a new edge to a tree creates a cycle (no longer a tree).
- **Removing an edge from a tree disconnects it (no longer a tree).**

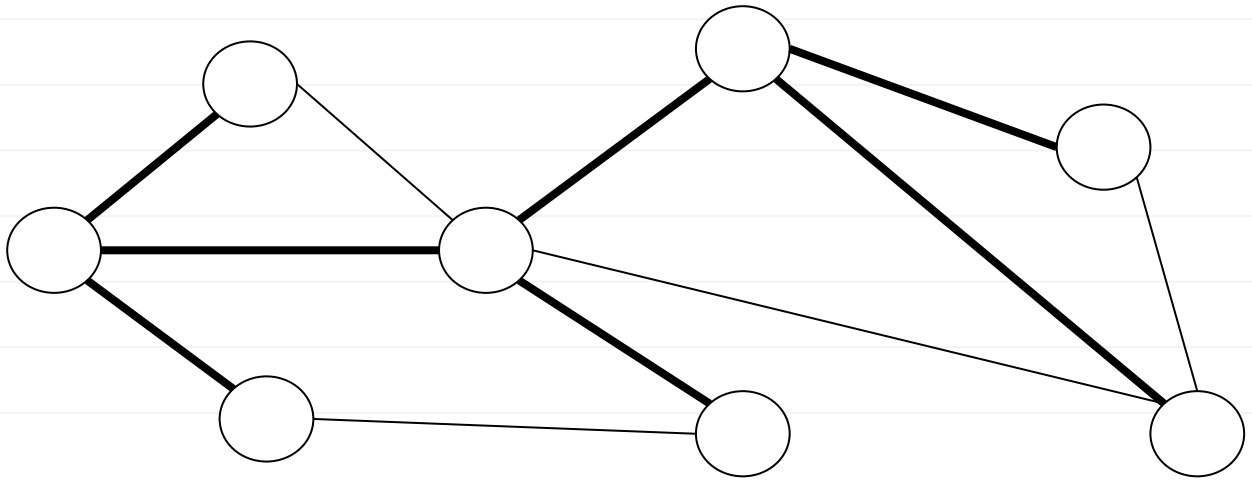
# Spanning Trees

- Let $G = (V , E)$ be a **connected** graph. A **spanning tree** of $G$ is a tree $T = (V , E_T)$ with the same nodes as $G$, and a *subset* of $G's$ edges.

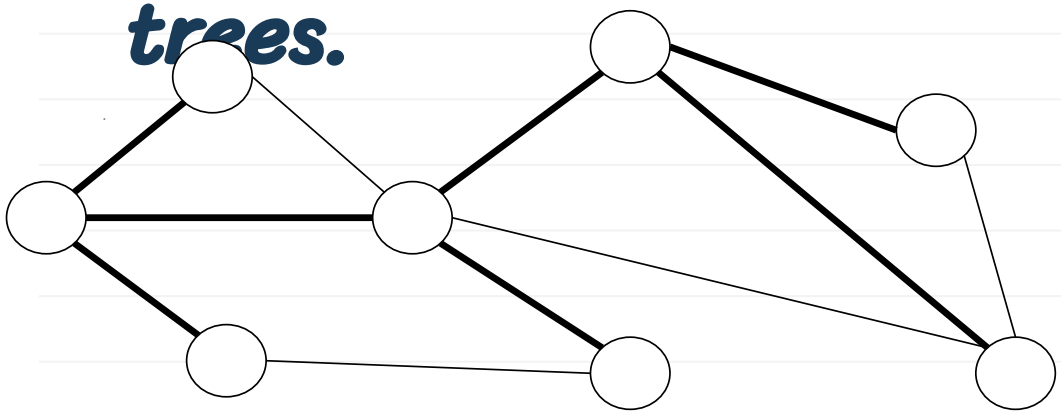


# Spanning Trees

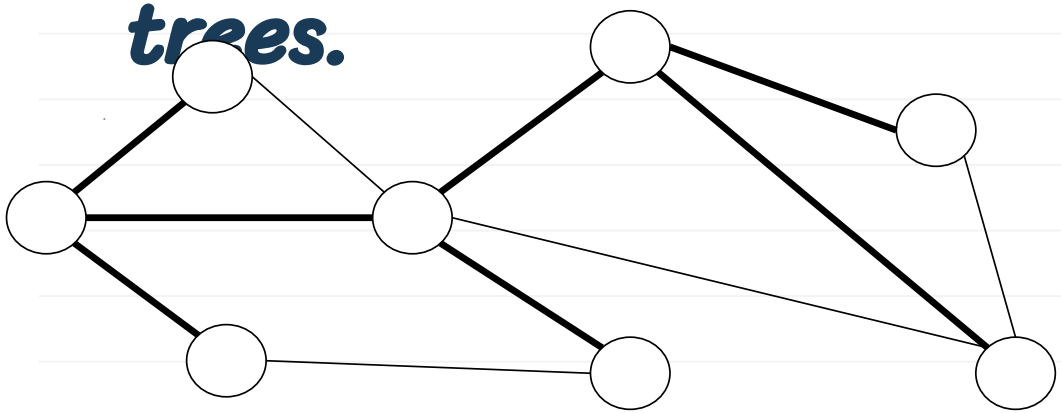
- Let $G = (V, E)$ be a **connected** graph. A **spanning tree** of $G$ is a tree $T = (V, E_T)$ with the same nodes as $G$, and a *subset* of $G's$ edges.

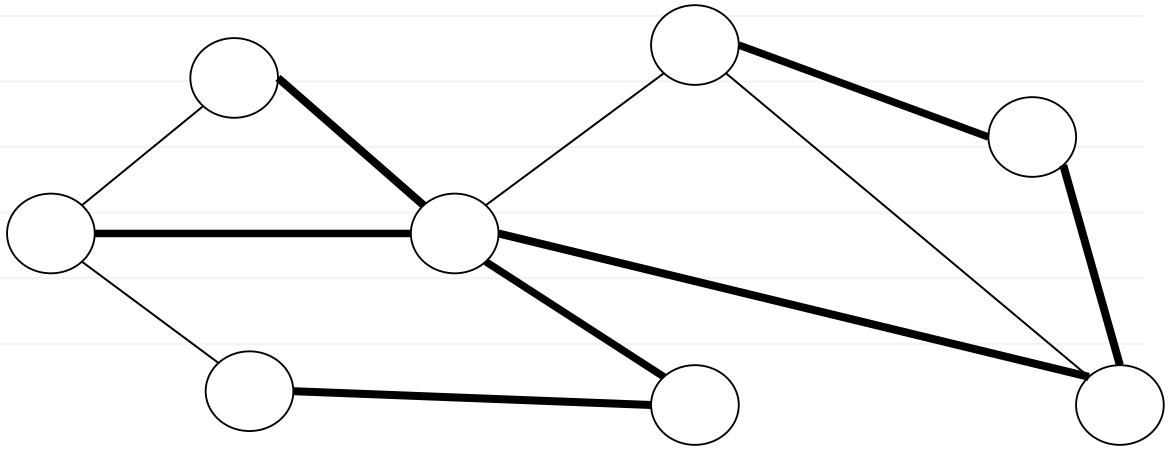


**The same graph can have many spanning trees.**



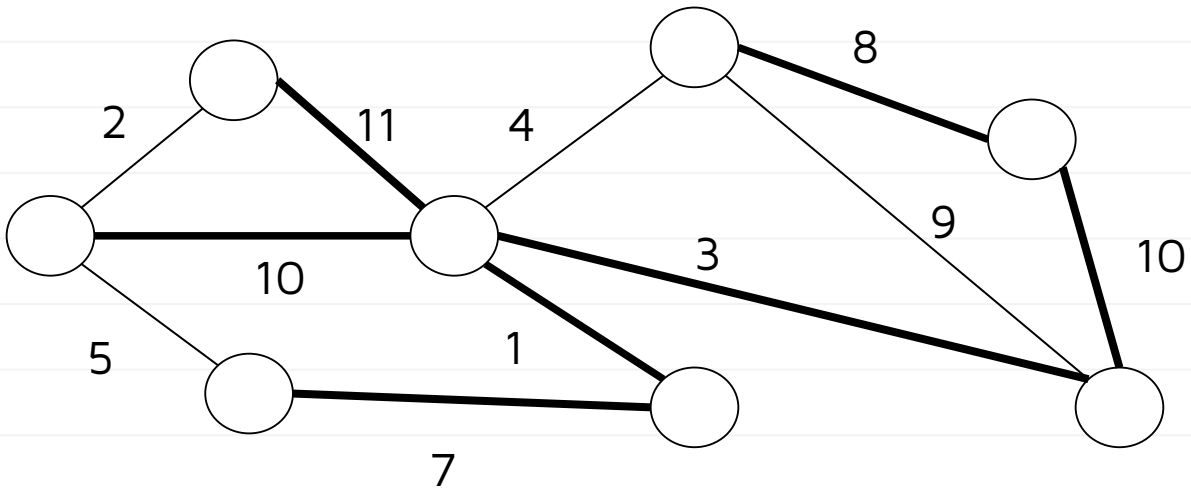
# The same graph can have many spanning trees.





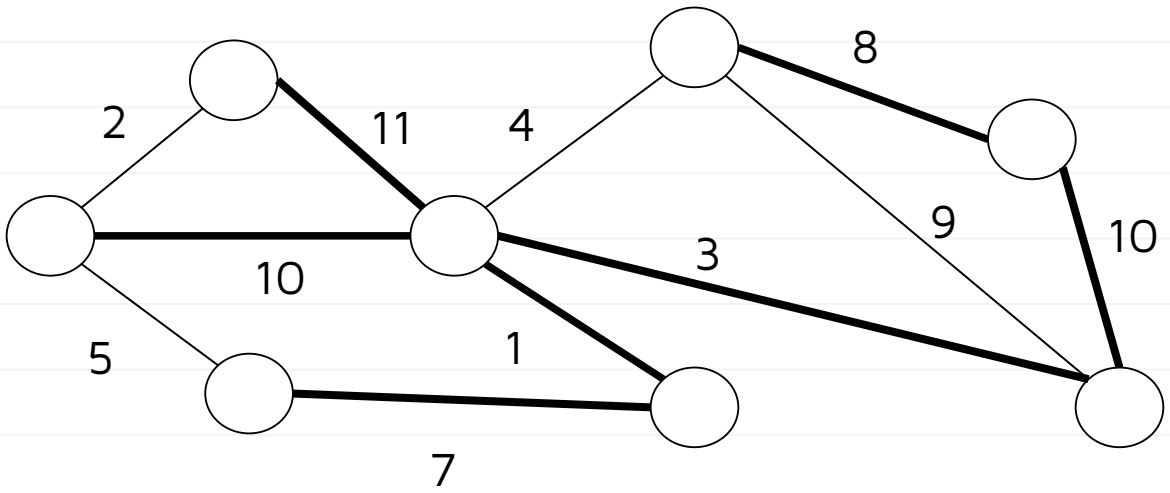
# Spanning Tree Cost

If $G = (V, E, \omega)$ is a weighted *undirected* graph, the **cost** (or **weight**) of a spanning tree is the total weight of the edges in the spanning tree.



# Spanning Tree Cost

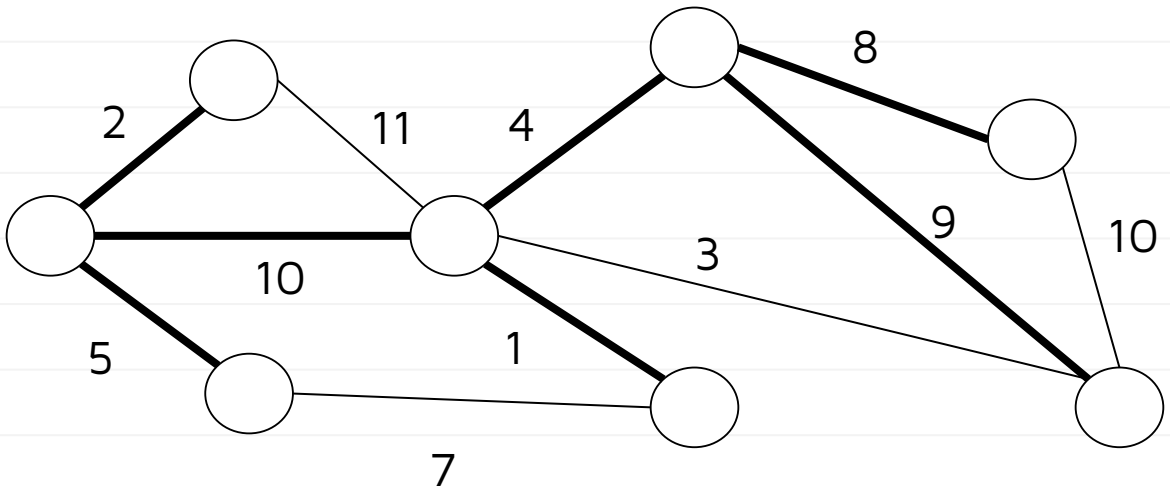
If $G = (V, E, \omega)$ is a weighted undirected graph, the **cost** (or **weight**) of a spanning tree is the total weight of the edges in the spanning tree.




**Cost**: 10 + 11 + 1 + 7 + 3 + 8 + 10 = 50

# Spanning Tree Cost

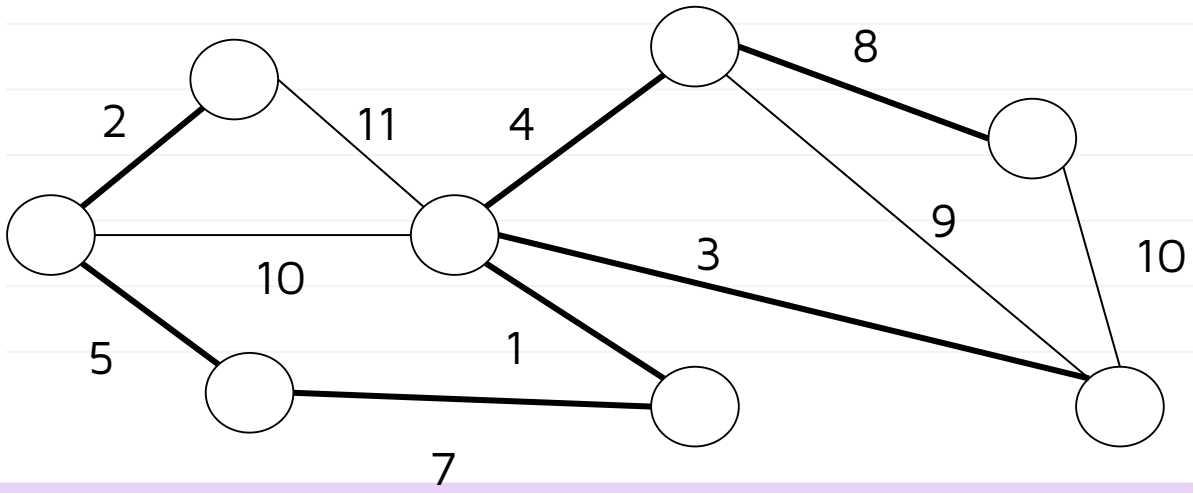
If $G = (V, E, \omega)$ is a weighted undirected graph, the **cost** (or **weight**) of a spanning tree is the total weight of the edges in the spanning tree.




**Cost**: 2 + 10 + 5 + 1 + 4 + 9 + 8 = 39

# Minimum Spanning Tree

- The **minimum spanning** tree problem is as follows:
  - **Given**: A weighted, undirected graph $G = (V, E, \omega)$.
  - **Compute**: a spanning tree of $G$ with minimum cost (i.e., minimum total edge weight).
- For a given graph, the MST may not be unique.




Cost:
1+2+3+4+5+7+8=30

# Question

Suppose the edges of a graph $G = (V, e, \omega)$ all have the same weight. How can we compute an MST of the graph?

# MSTs in Data Science?

- Do we need to find MSTs in data science?
- Actually, yes! (Next lecture)

# Prim's Algorithm

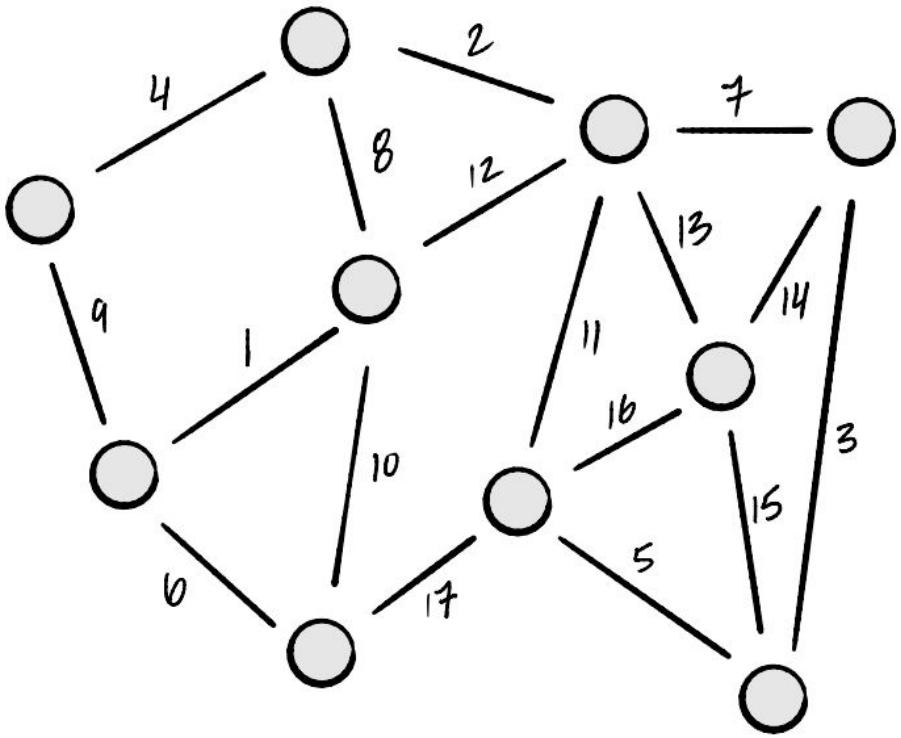
# Building MSTs

- How do we build a MST efficiently?
- We'll adopt a **greedy** approach.
  - Build a tree edge-by-edge.
  - At every step, doing what looks best at the moment.
- This strategy isn't guaranteed to work in all of life's situations, but it works for building MSTs.

# Two Greedy Approaches

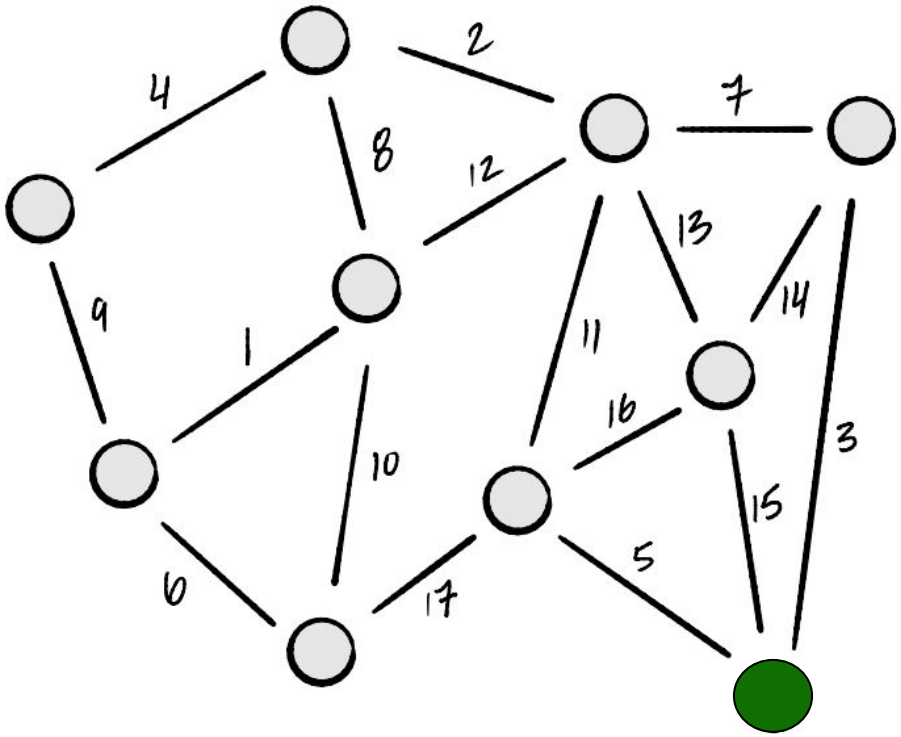
- We'll look at two greedy algorithms:
  - **Today**: *Prim's Algorithm*
  - Next time: Kruskal's Algorithm
- Differ in the order in which edges are added to tree.
- Also differ in time complexity.

# Prim's Algorithm, Informally



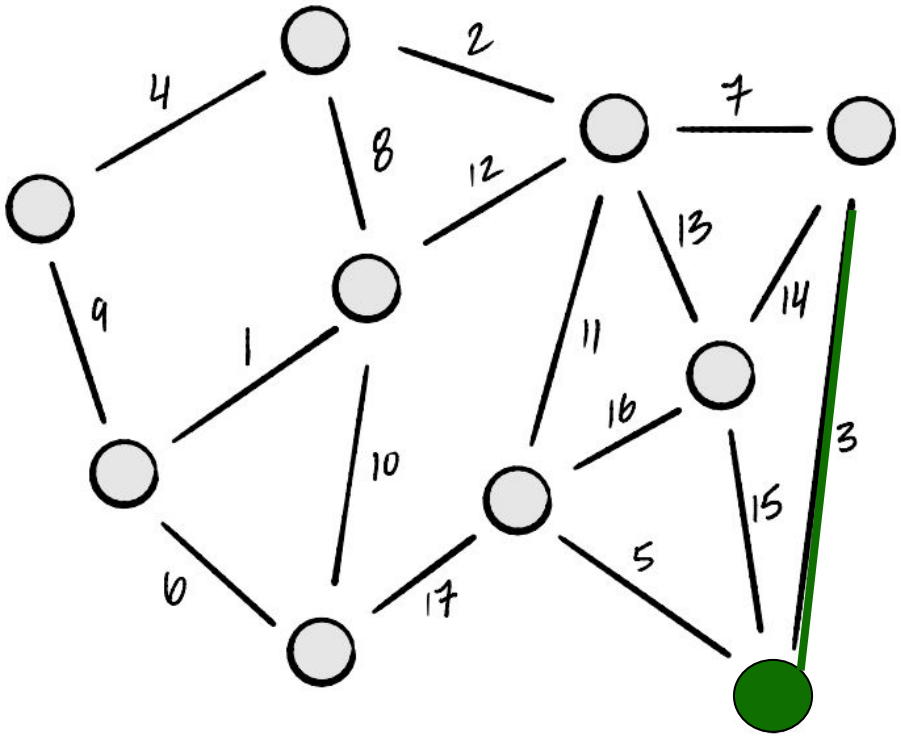

- Start by picking any node to add to "tree", $T$.
- While $T$ is not a spanning tree, greedily add lightest edge from a node in $T$ to a node not in $T$.
  - "Lightest" = edge with the smallest weight.

# Prim's Algorithm, Informally



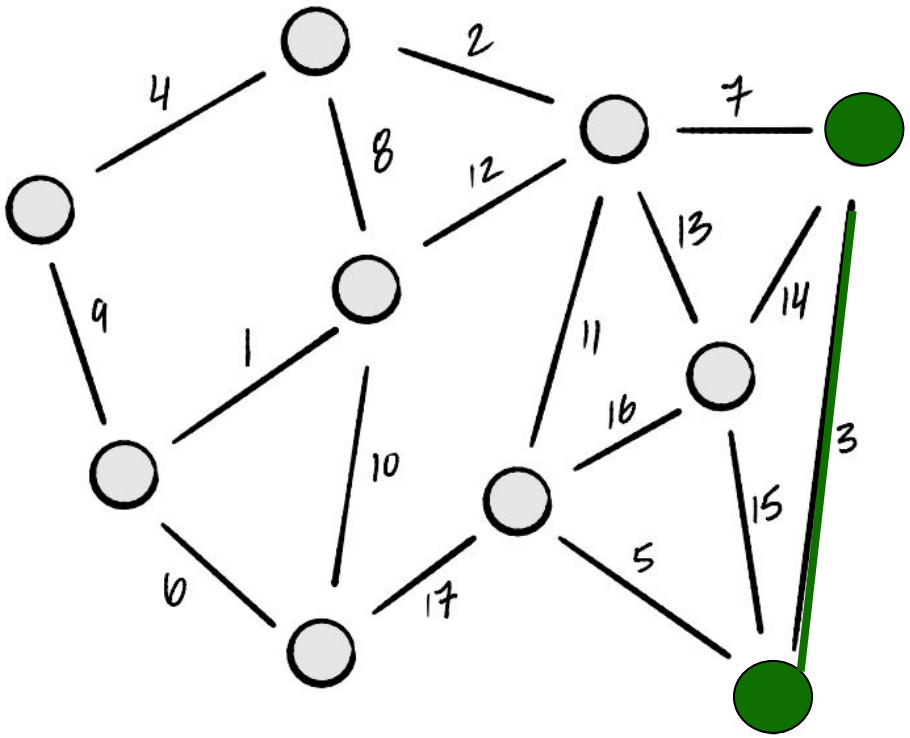

- Start by picking any node to add to "tree", $T$.
- While $T$ is not a spanning tree, greedily add **lightest** edge from a node in $T$ to a node not in $T$.
  - "Lightest" = edge with the smallest weight.

# Prim's Algorithm, Informally



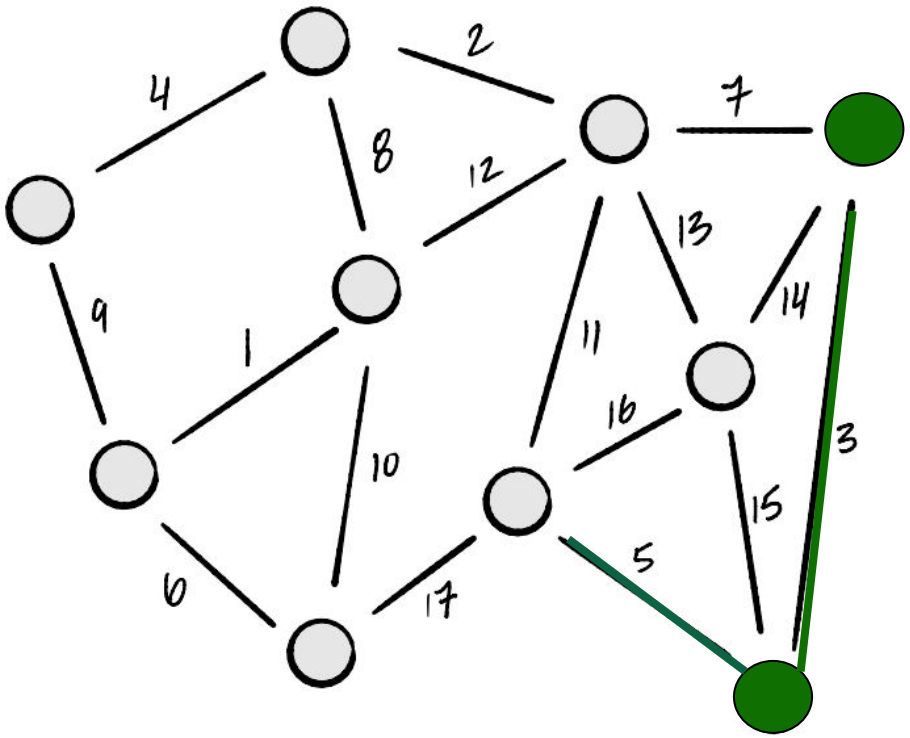

- Start by picking any node to add to "tree", $T$.
- While $T$ is not a spanning tree, greedily add **lightest** edge from a node in $T$ to a node not in $T$.
  - "Lightest" = edge with the smallest weight.

# Prim's Algorithm, Informally



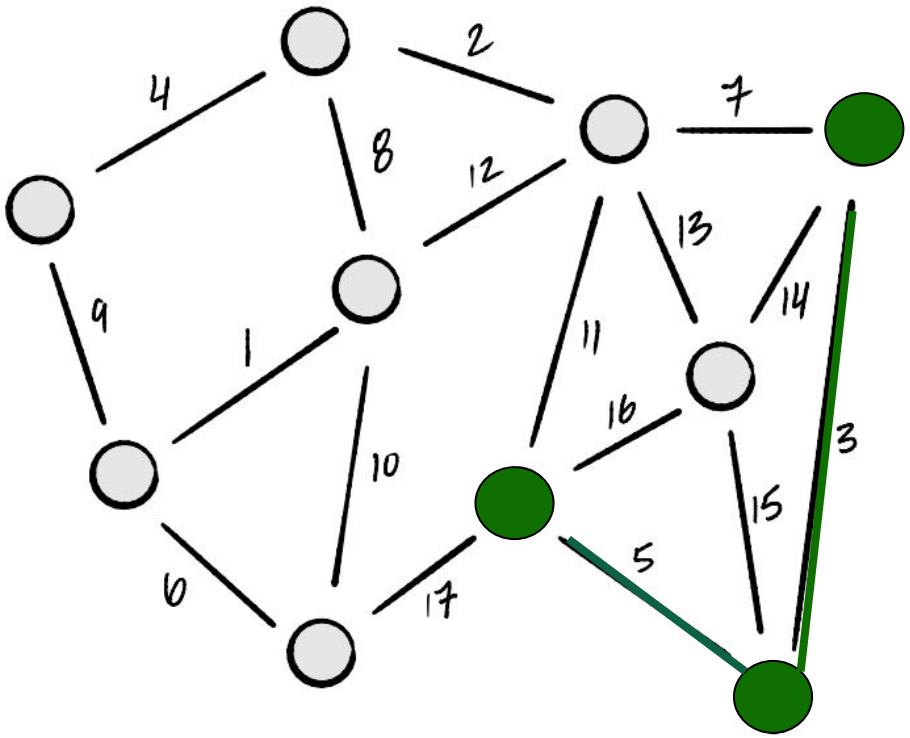

- Start by picking any node to add to "tree", $T$.
- While $T$ is not a spanning tree, greedily add **lightest** edge from a node in $T$ to a node not in $T$.
  - "Lightest" = edge with the smallest weight.

# Prim's Algorithm, Informally



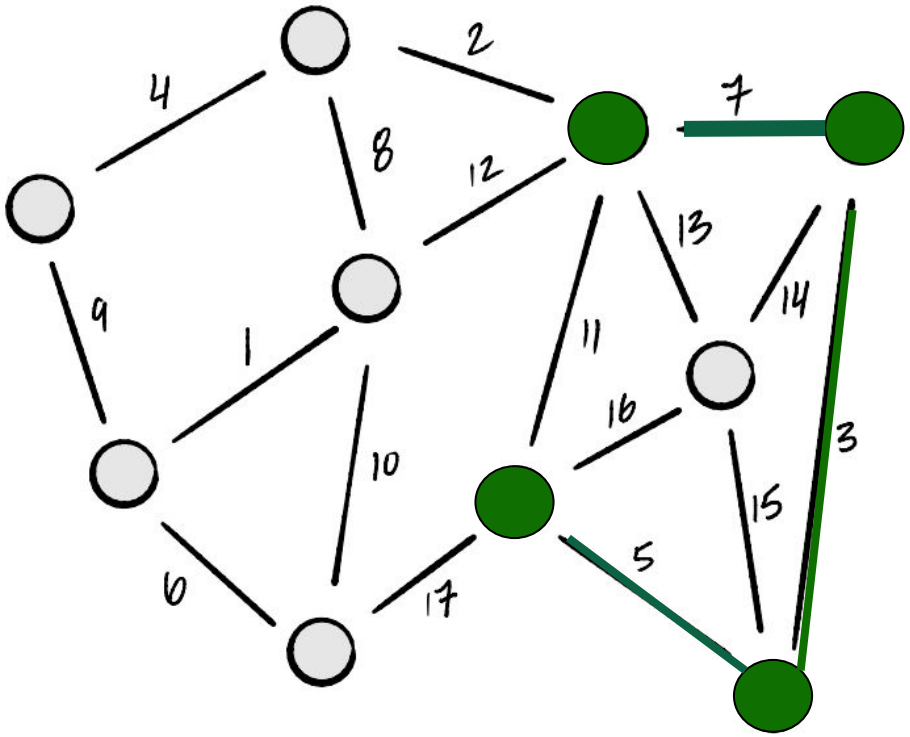

- Start by picking any node to add to "tree", $T$.
- While $T$ is not a spanning tree, greedily add **lightest** edge from a node in $T$ to a node not in $T$.
  - "Lightest" = edge with the smallest weight.

# Prim's Algorithm, Informally



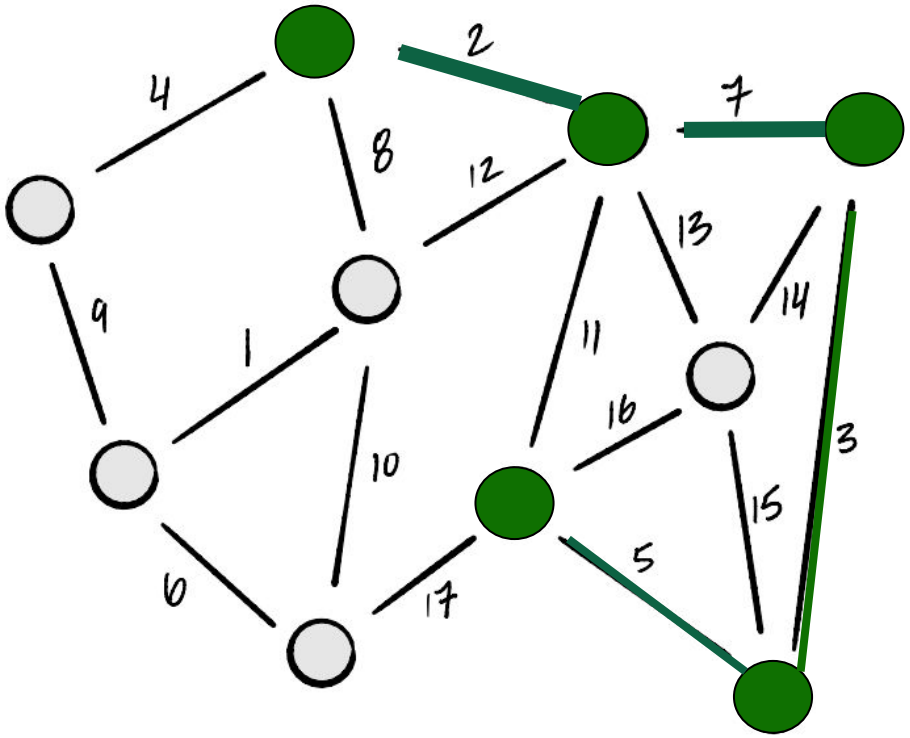

- Start by picking any node to add to "tree", $T$.
- While $T$ is not a spanning tree, greedily add **lightest** edge from a node in $T$ to a node not in $T$.
  - "Lightest" = edge with the smallest weight.

# Prim's Algorithm, Informally



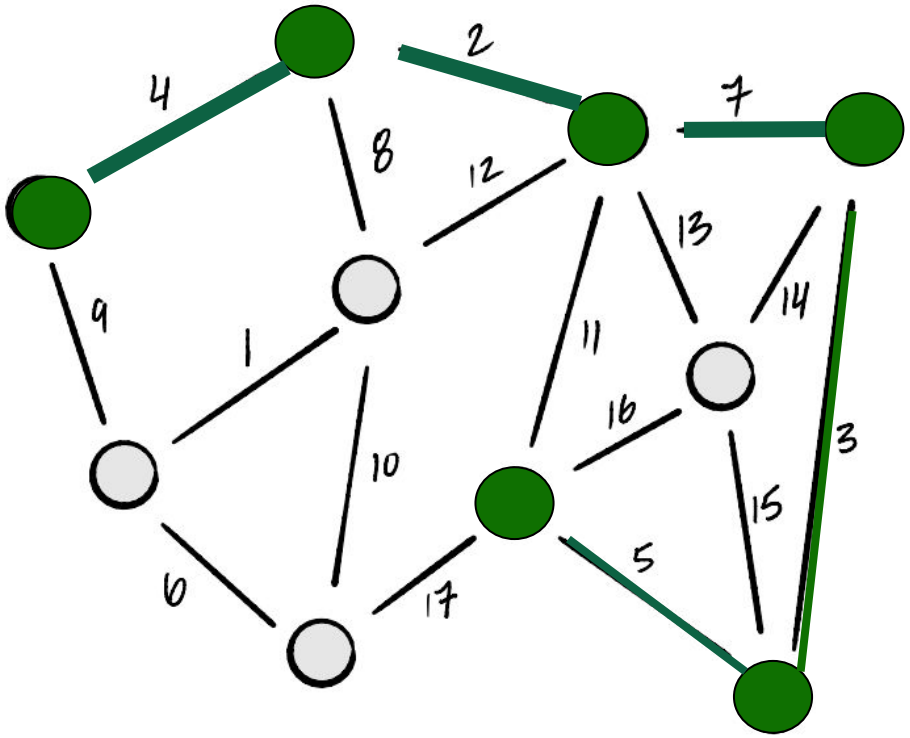

- Start by picking any node to add to "tree", $T$.
- While $T$ is not a spanning tree, greedily add **lightest** edge from a node in $T$ to a node not in $T$.
  - "Lightest" = edge with the smallest weight.

# Prim's Algorithm, Informally



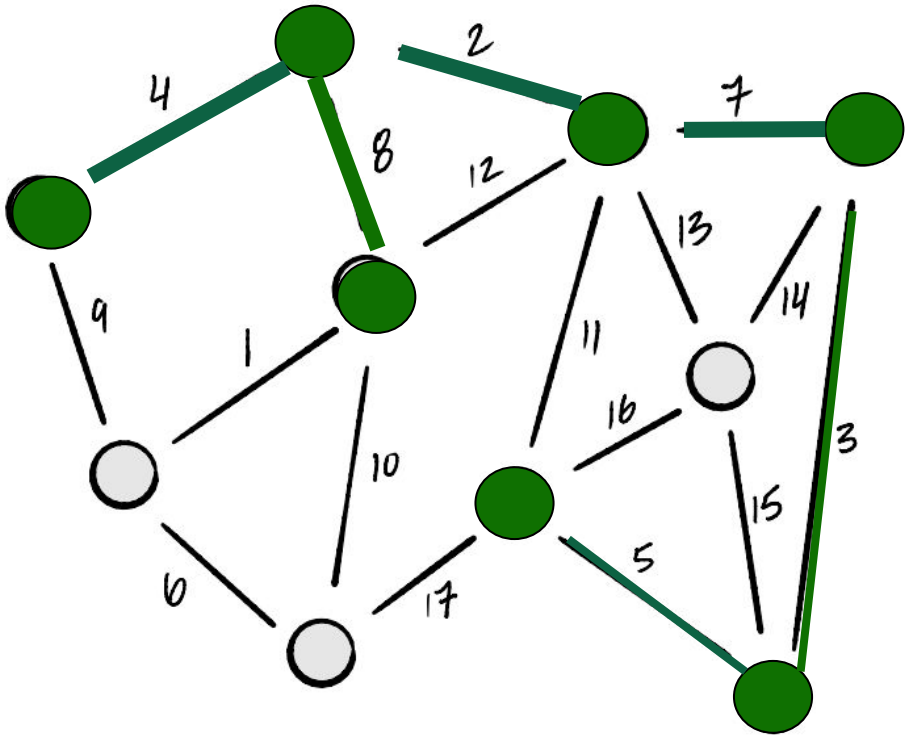

- Start by picking any node to add to "tree", $T$.
- While $T$ is not a spanning tree, greedily add **lightest** edge from a node in $T$ to a node not in $T$.
  - "Lightest" = edge with the smallest weight.

# Prim's Algorithm, Informally



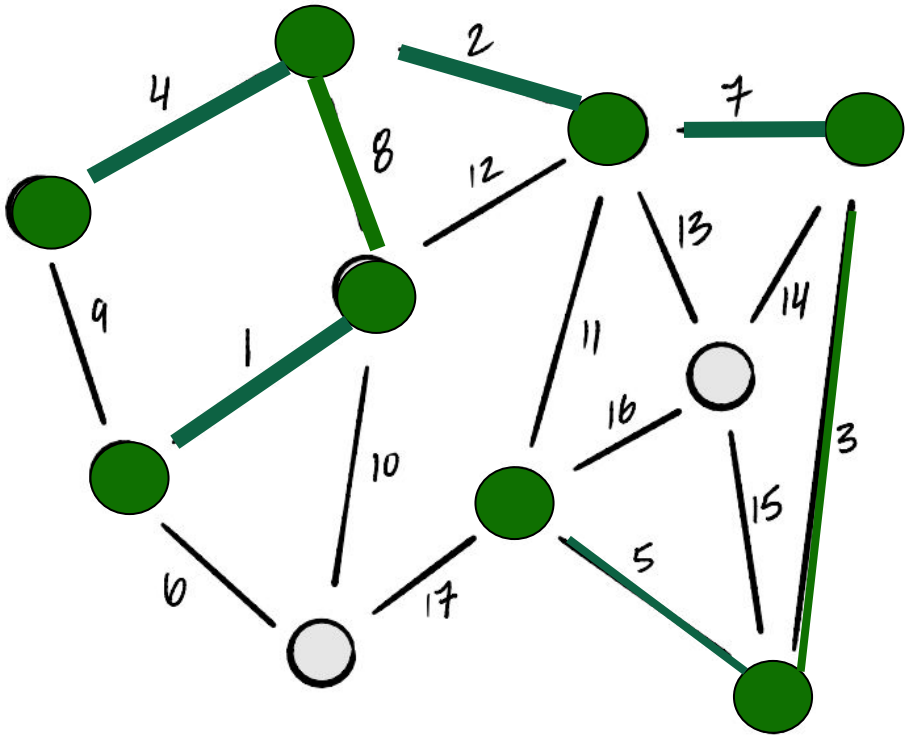

- Start by picking any node to add to "tree", $T$.
- While $T$ is not a spanning tree, greedily add **lightest** edge from a node in $T$ to a node not in $T$.
  - "Lightest" = edge with the smallest weight.

# Prim's Algorithm, Informally



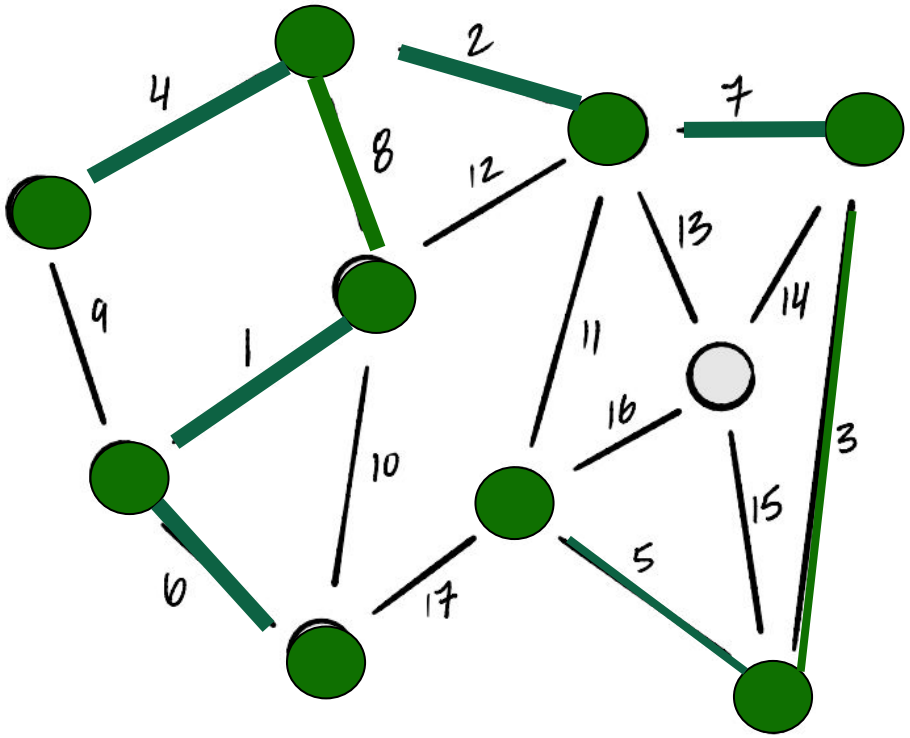

- Start by picking any node to add to "tree", $T$.
- While $T$ is not a spanning tree, greedily add **lightest** edge from a node in $T$ to a node not in $T$.
  - "Lightest" = edge with the smallest weight.

# Prim's Algorithm, Informally



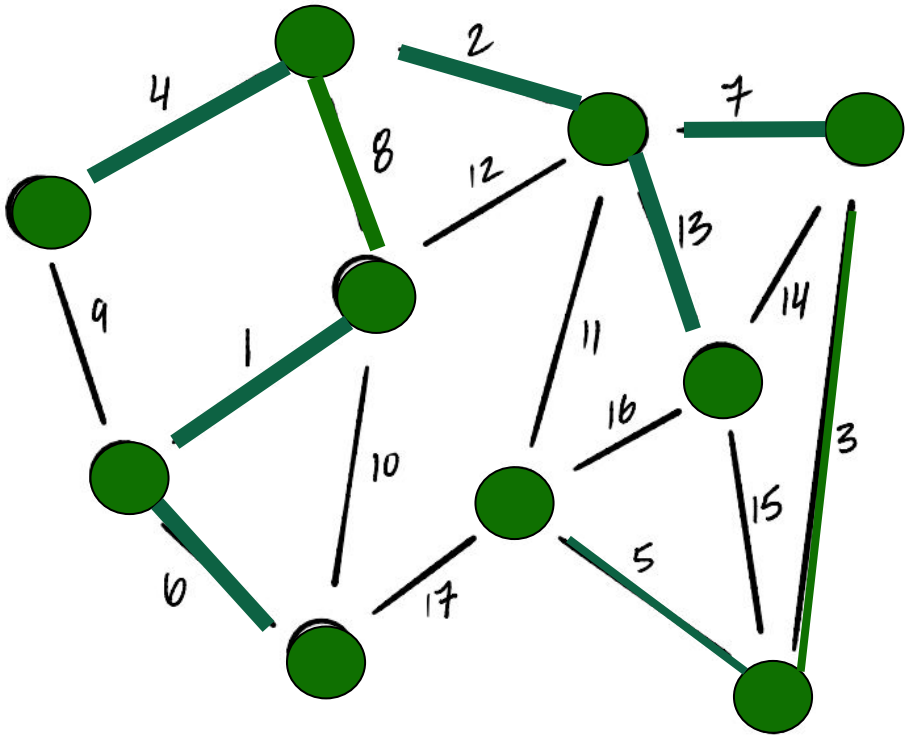

- Start by picking any node to add to "tree", $T$.
- While $T$ is not a spanning tree, greedily add **lightest** edge from a node in $T$ to a node not in $T$.
  - "Lightest" = edge with the smallest weight.

# Prim's Algorithm, Informally



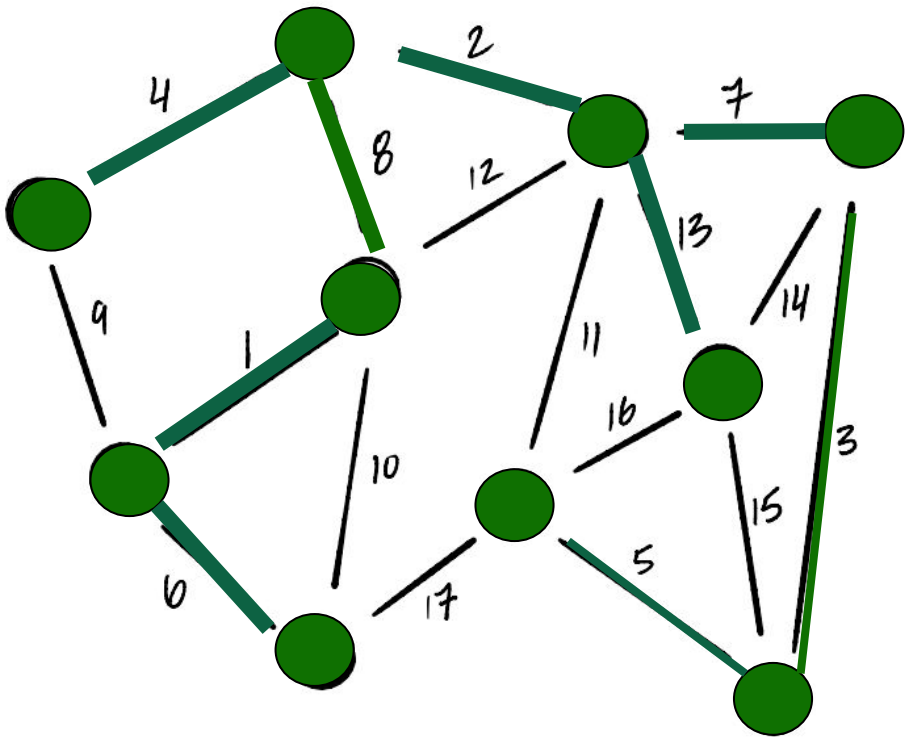

- Start by picking any node to add to "tree", $T$.
- While $T$ is not a spanning tree, greedily add **lightest** edge from a node in $T$ to a node not in $T$.
  - "Lightest" = edge with the smallest weight.

# Prim's Algorithm, Informally




- Start by picking any node to add to "tree", $T$.
- While $T$ is not a spanning tree, greedily add **lightest** edge from a node in $T$ to a node not in $T$.
  - "Lightest" = edge with the smallest weight.

# Prim's Algorithm, Informally

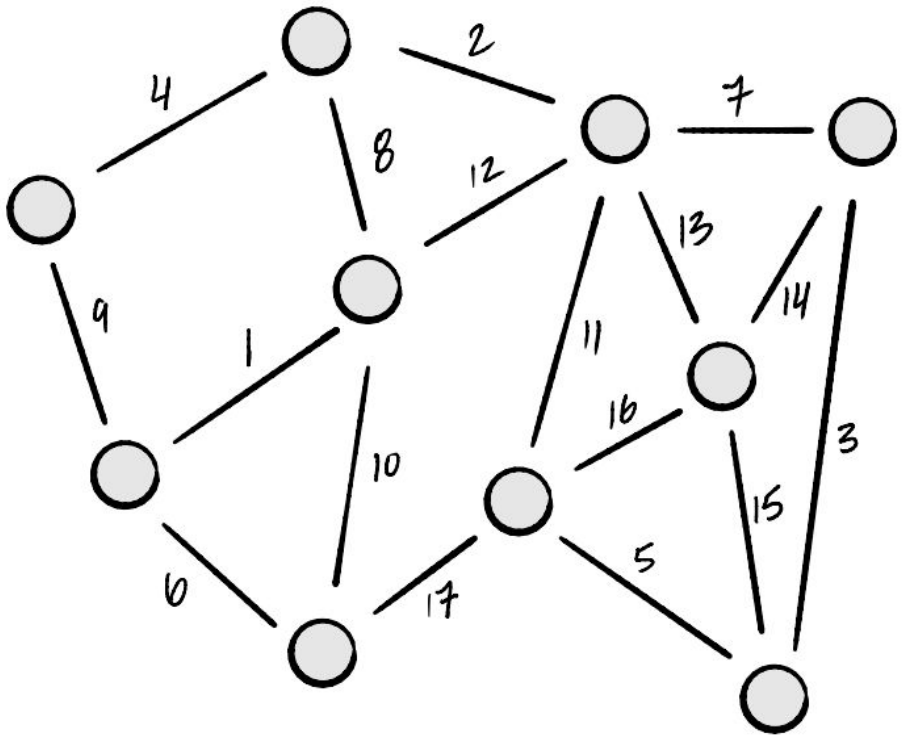



- Start by picking any node to add to "tree", $T$.
- While $T$ is not a spanning tree, greedily add **lightest** edge from a node in $T$ to a node not in $T$.
  - "Lightest" = edge with the smallest weight.
- **Is this guaranteed to work?** Yes, as we'll see.

# Prim's Algorithm, Equivalently

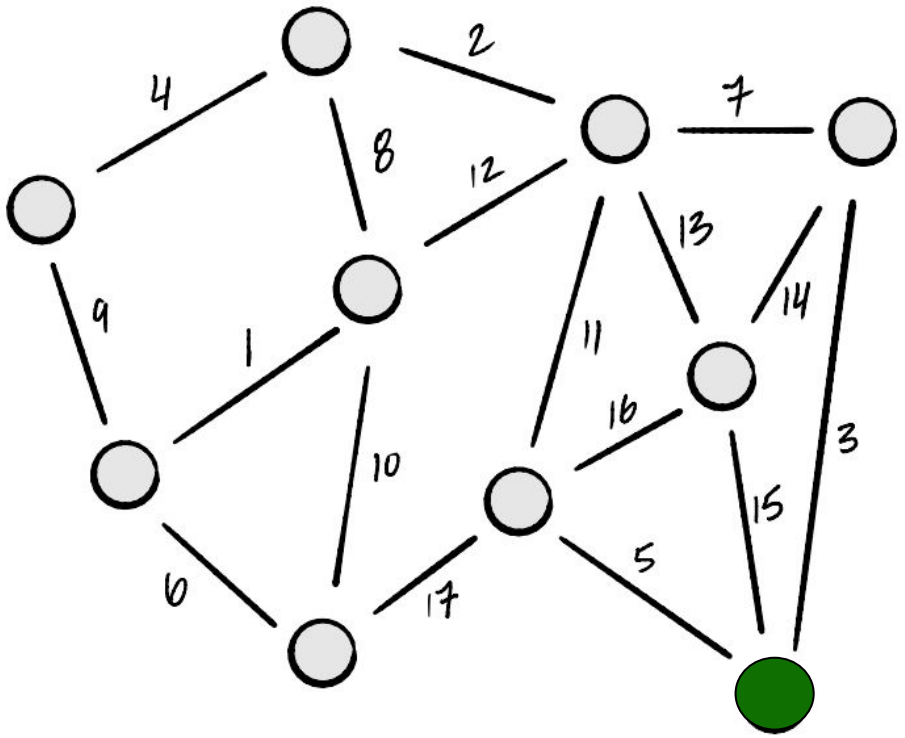
- **For each node $u$, store:**
  - estimated cost of adding node to tree;
  - estimated "predecessor" $v$ in the tree.
- **At each step,**
  - Find node with *smallest* estimated cost.
  - Add to tree $T$ by including edge with estimated "predecessor".
  - Update cost of neighbors.
- Same as adding lightest edge from $T$ to outside $T$ at every step!

# Prim's Algorithm, Equivalently



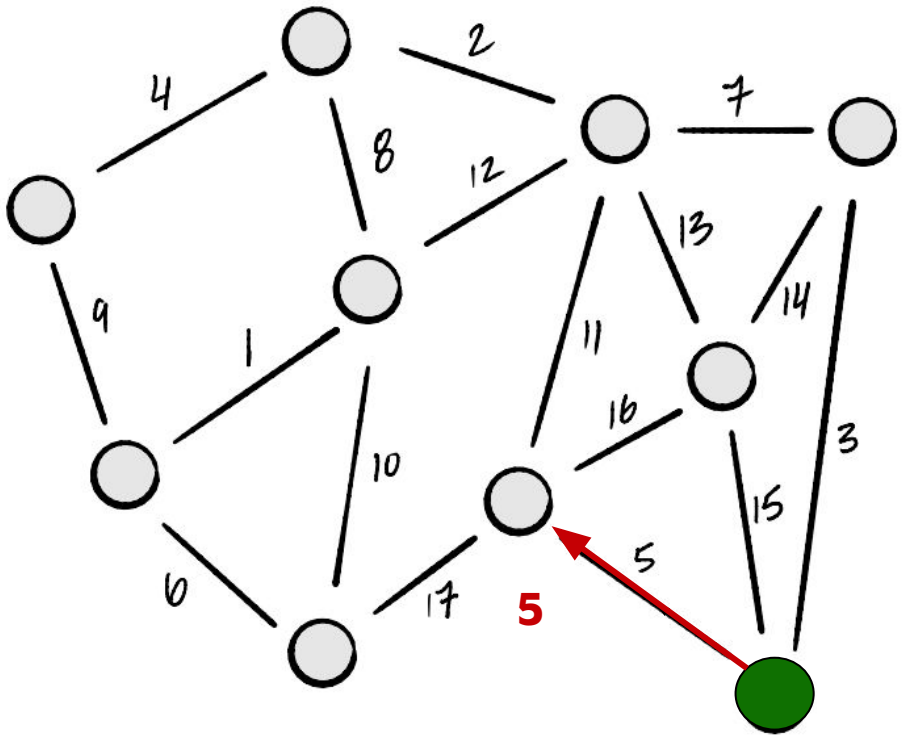

- While $T$ is not a tree:
  - find the node $u \notin T$ with smallest cost.
  - add the edge between $u$ and its estimated "predecessor" to $T$.
  - update estimated cost/pred. of $u$'s neighbors which are not already in the tree.

# Prim's Algorithm, Equivalently



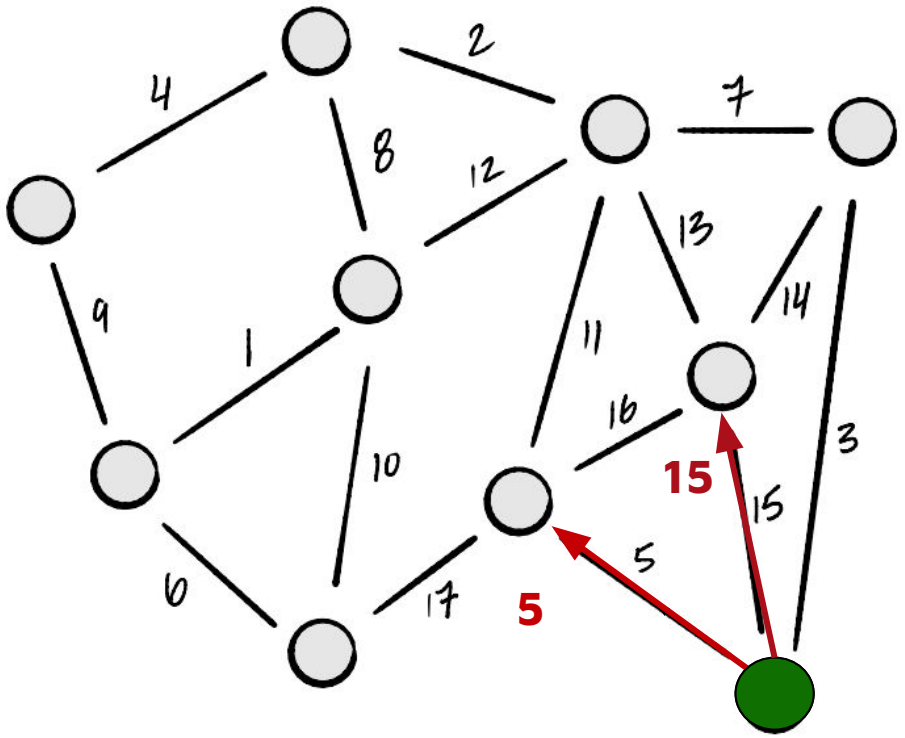

- While $T$ is not a tree:
  - find the node $u \notin T$ with smallest cost.
  - add the edge between $u$ and its estimated "predecessor" to $T$.
  - update estimated cost/pred. of $u$'s neighbors which are not already in the tree.

# Prim's Algorithm, Equivalently



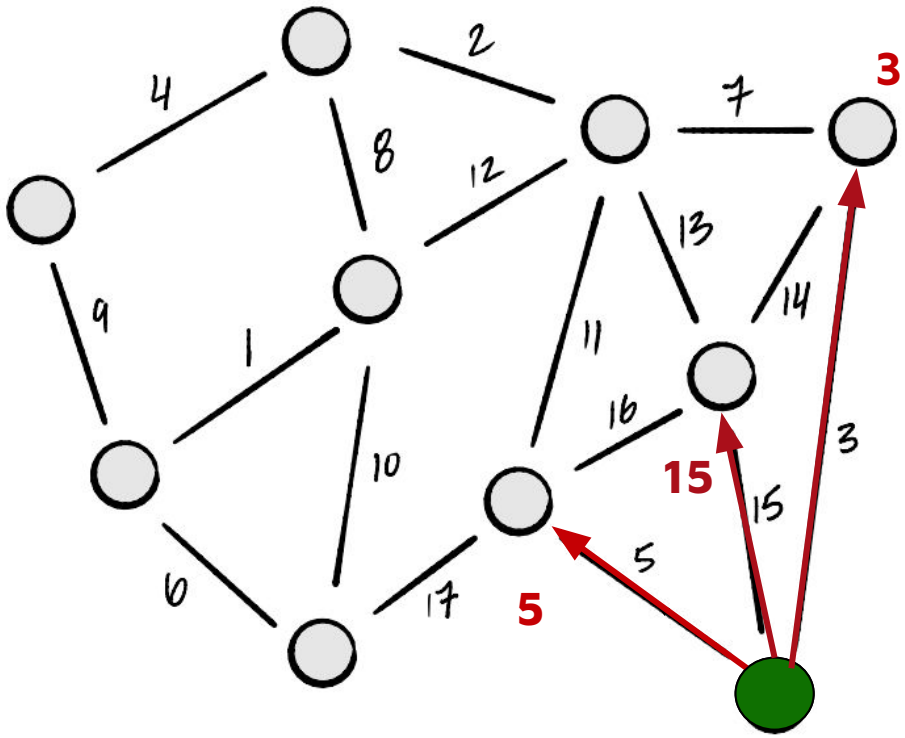

- While $T$ is not a tree:
  - find the node $u \notin T$ with smallest cost.
  - add the edge between $u$ and its estimated "predecessor" to $T$.
  - update estimated cost/pred. of $u$'s neighbors which are not already in the tree.

# Prim's Algorithm, Equivalently



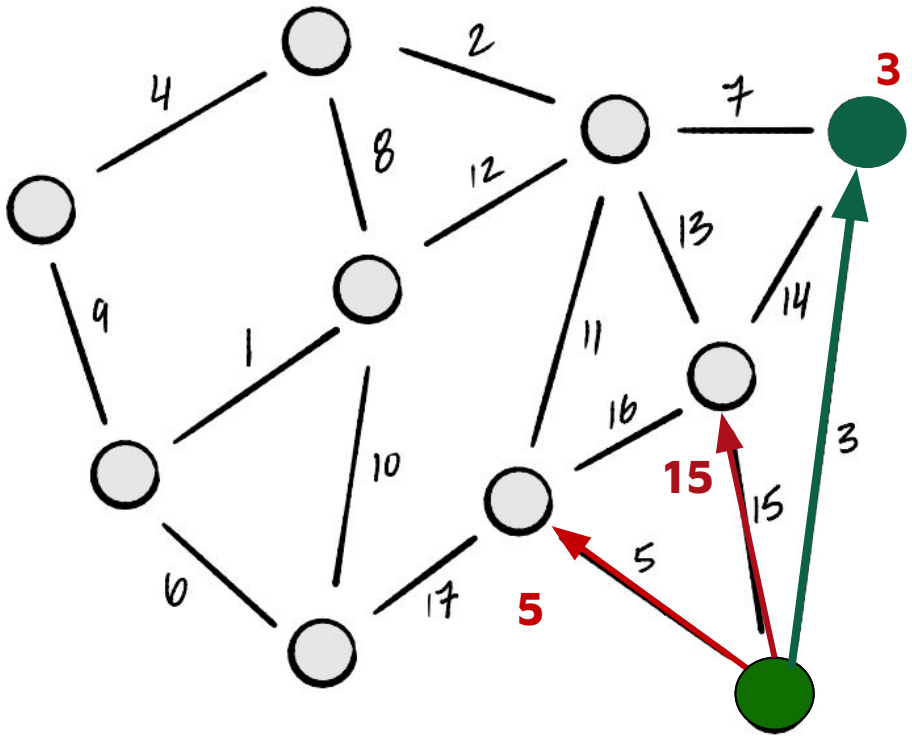

- While $T$ is not a tree:
  - find the node $u \notin T$ with smallest cost.
  - add the edge between $u$ and its estimated "predecessor" to $T$.
  - update estimated cost/pred. of $u$'s neighbors which are not already in the tree.

# Prim's Algorithm, Equivalently



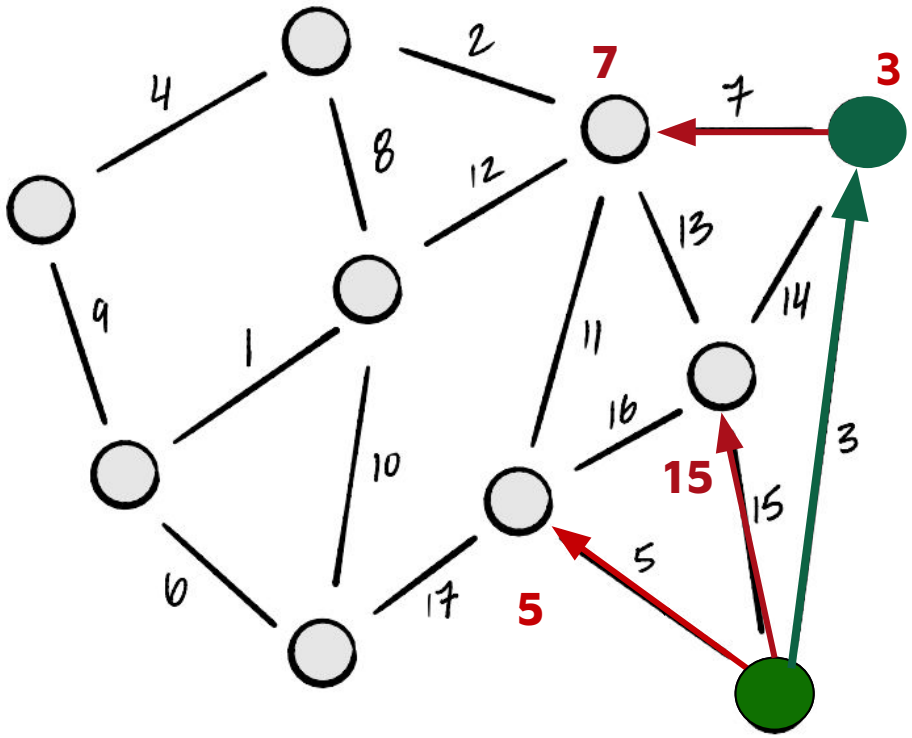

- While $T$ is not a tree:
  - find the node $u \notin T$ with smallest cost.
  - add the edge between $u$ and its estimated "predecessor" to $T$.
  - update estimated cost/pred. of $u$'s neighbors which are not already in the tree.

# Prim's Algorithm, Equivalently



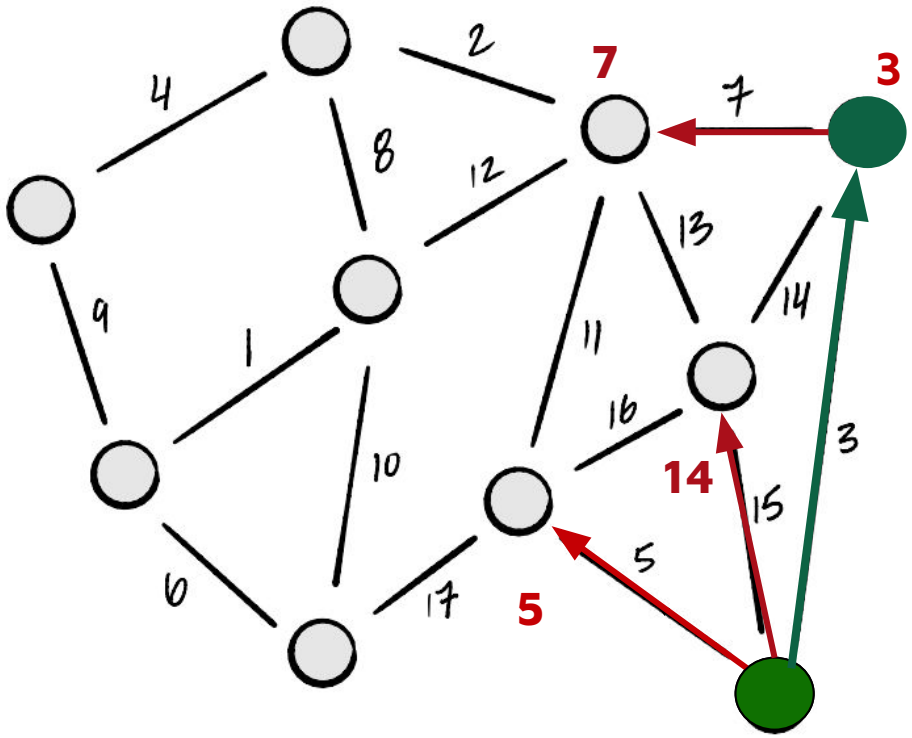

- While $T$ is not a tree:
  - find the node $u \notin T$ with smallest cost.
  - add the edge between $u$ and its estimated "predecessor" to $T$.
  - update estimated cost/pred. of $u$'s neighbors which are not already in the tree.

# Prim's Algorithm, Equivalently



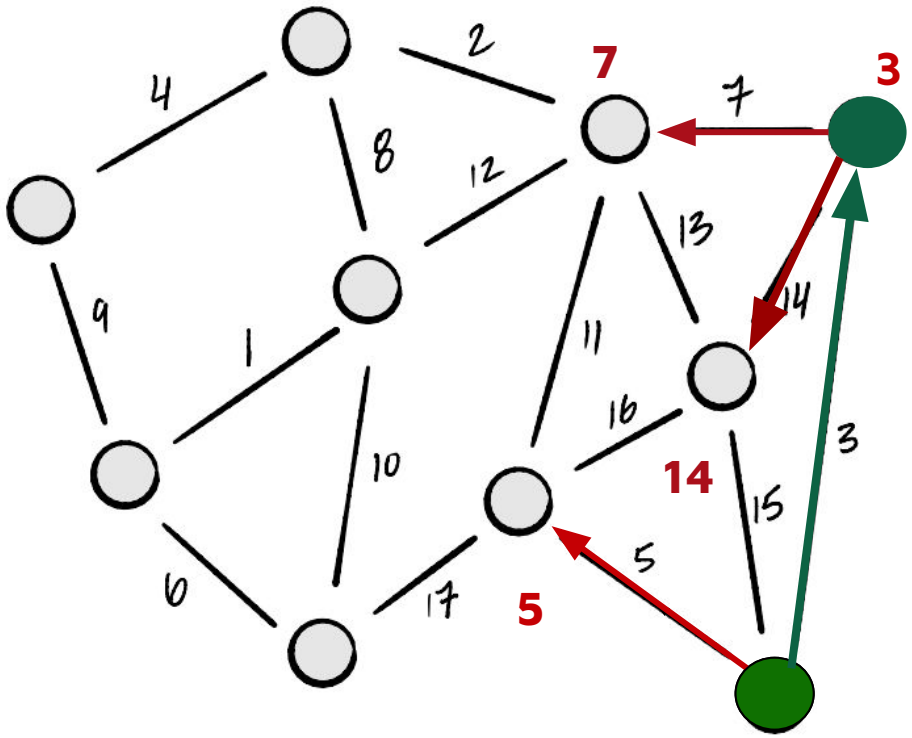

- While $T$ is not a tree:
  - find the node $u \notin T$ with smallest cost.
  - add the edge between $u$ and its estimated "predecessor" to $T$.
  - update estimated cost/pred. of $u$'s neighbors which are not already in the tree.

# Prim's Algorithm, Equivalently



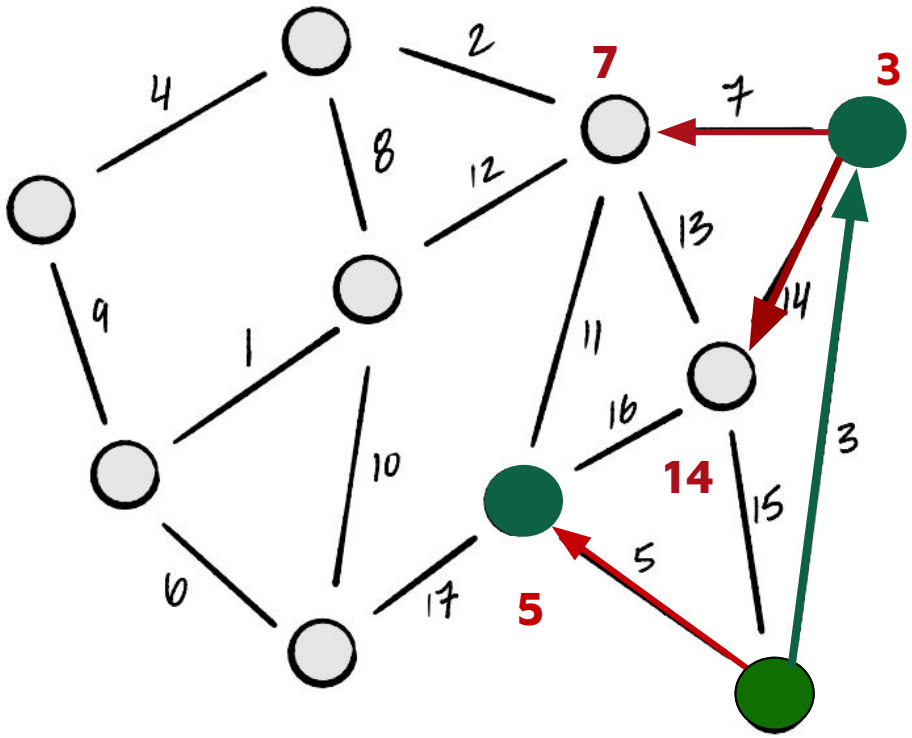

- While $T$ is not a tree:
  - find the node $u \notin T$ with smallest cost.
  - add the edge between $u$ and its estimated "predecessor" to $T$.
  - update estimated cost/pred. of $u$'s neighbors which are not already in the tree.

# Prim's Algorithm, Equivalently



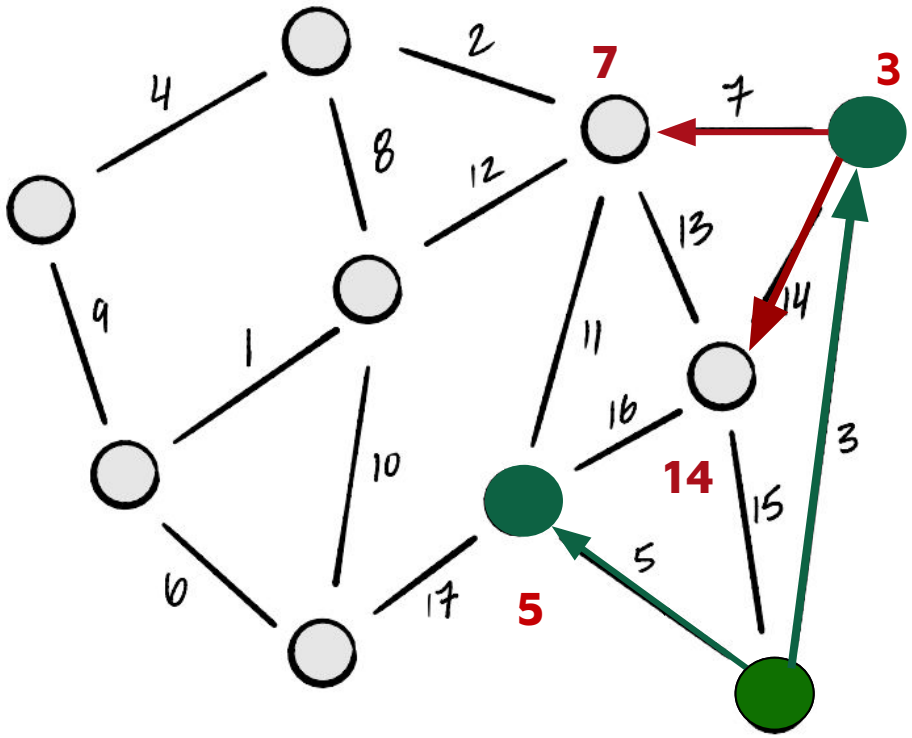

- While $T$ is not a tree:
  - find the node $u \notin T$ with smallest cost.
  - add the edge between $u$ and its estimated "predecessor" to $T$.
  - update estimated cost/pred. of $u$'s neighbors which are not already in the tree.

# Prim's Algorithm, Equivalently



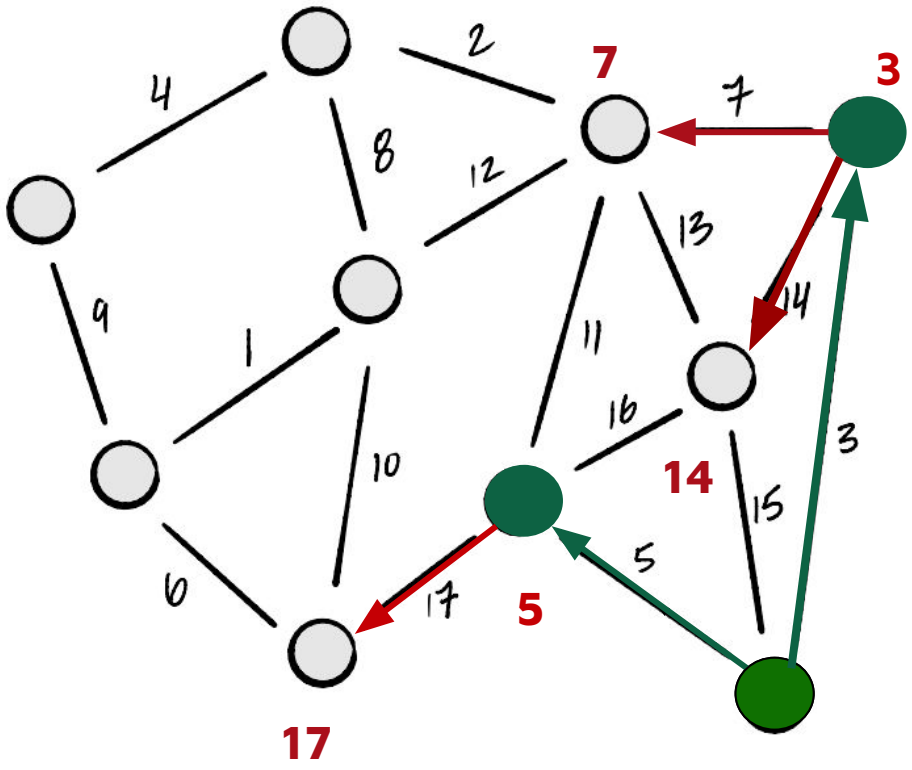

- While $T$ is not a tree:
  - find the node $u \notin T$ with smallest cost.
  - add the edge between $u$ and its estimated "predecessor" to $T$.
  - update estimated cost/pred. of $u$'s neighbors which are not already in the tree.

# Prim's Algorithm, Equivalently



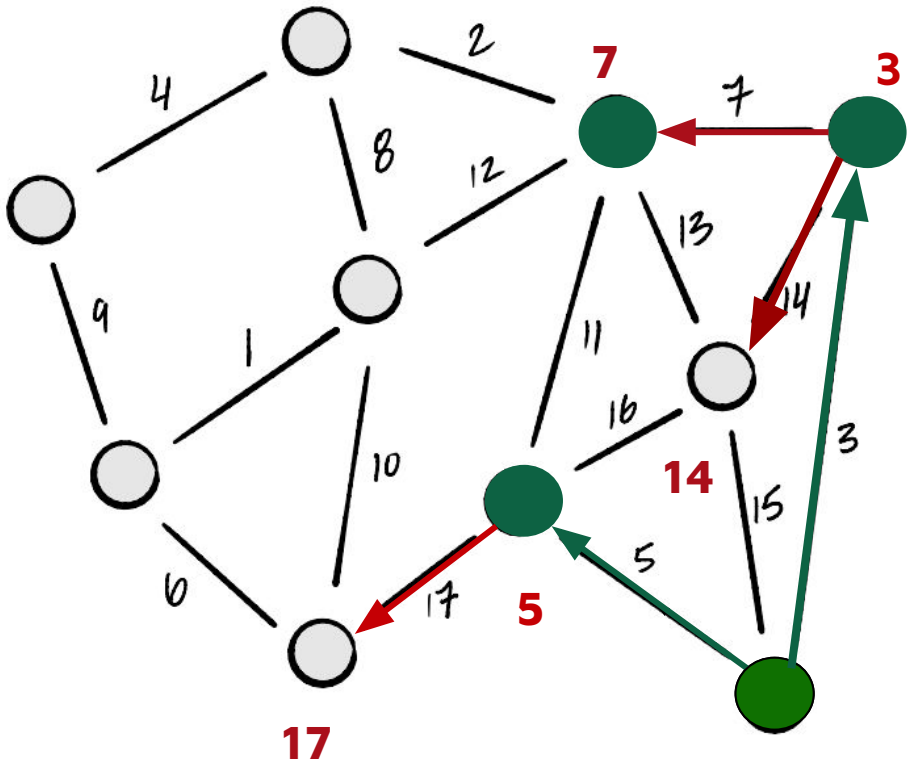

- While $T$ is not a tree:
  - find the node $u \notin T$ with smallest cost.
  - add the edge between $u$ and its estimated "predecessor" to $T$.
  - update estimated cost/pred. of $u$'s neighbors which are not already in the tree.

# Prim's Algorithm, Equivalently



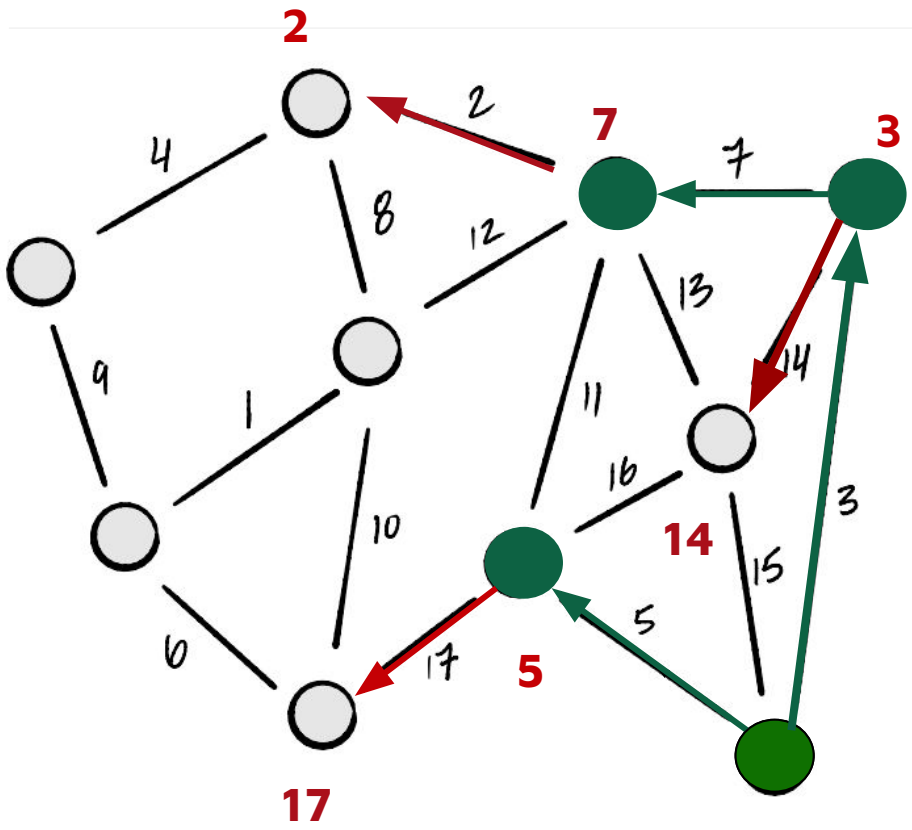

- While $T$ is not a tree:
  - find the node $u \notin T$ with smallest cost.
  - add the edge between $u$ and its estimated "predecessor" to $T$.
  - update estimated cost/pred. of $u$'s neighbors which are not already in the tree.

# Prim's Algorithm, Equivalently



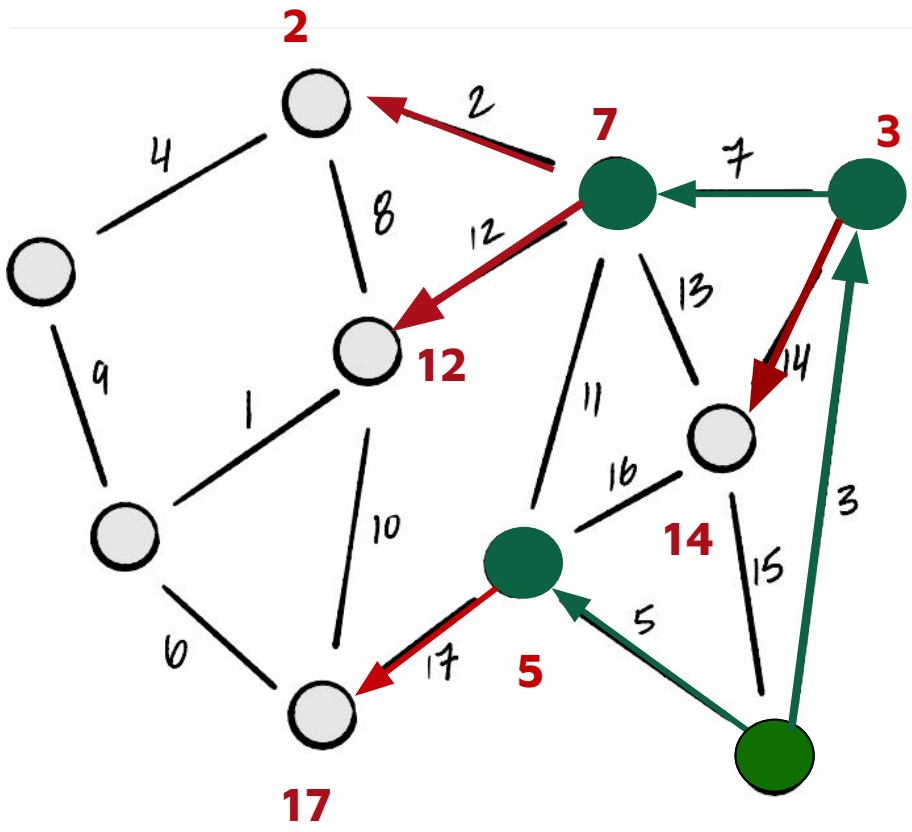

- While $T$ is not a tree:
  - find the node $u \notin T$ with smallest cost.
  - add the edge between $u$ and its estimated "predecessor" to $T$.
  - update estimated cost/pred. of $u$'s neighbors which are not already in the tree.

# Prim's Algorithm, Equivalently



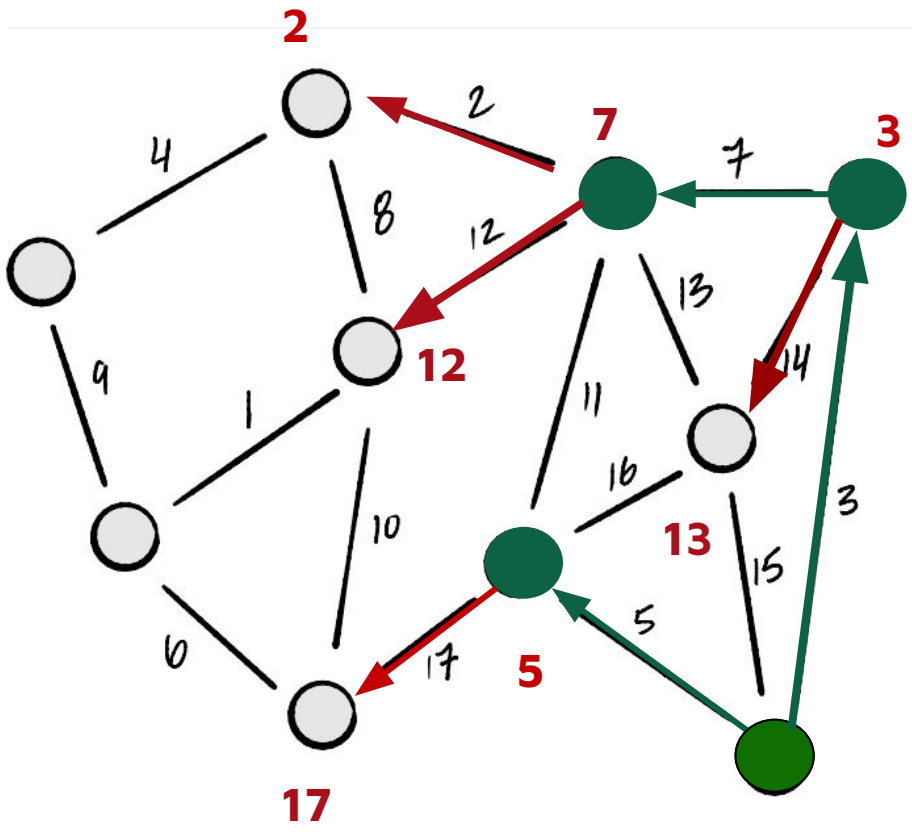

- While $T$ is not a tree:
  - find the node $u \notin T$ with smallest cost.
  - add the edge between $u$ and its estimated "predecessor" to $T$.
  - update estimated cost/pred. of $u$'s neighbors which are not already in the tree.

# Prim's Algorithm, Equivalently



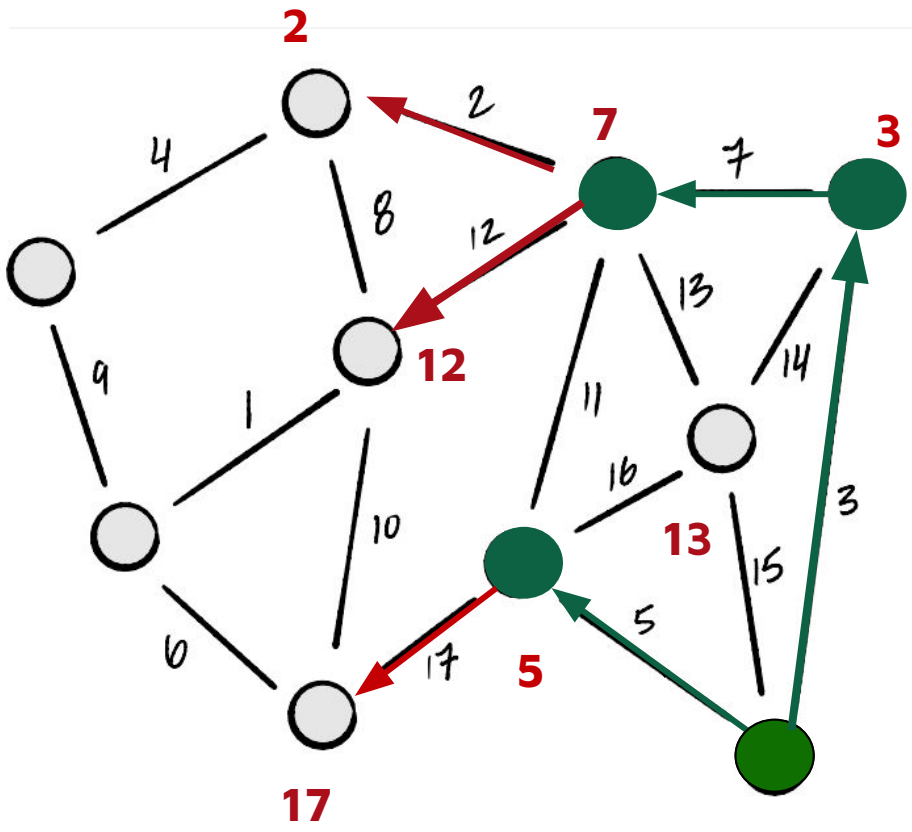

- While $T$ is not a tree:
  - find the node $u \notin T$ with smallest cost.
  - add the edge between $u$ and its estimated "predecessor" to $T$.
  - update estimated cost/pred. of $u$'s neighbors which are not already in the tree.

# Prim's Algorithm, Equivalently



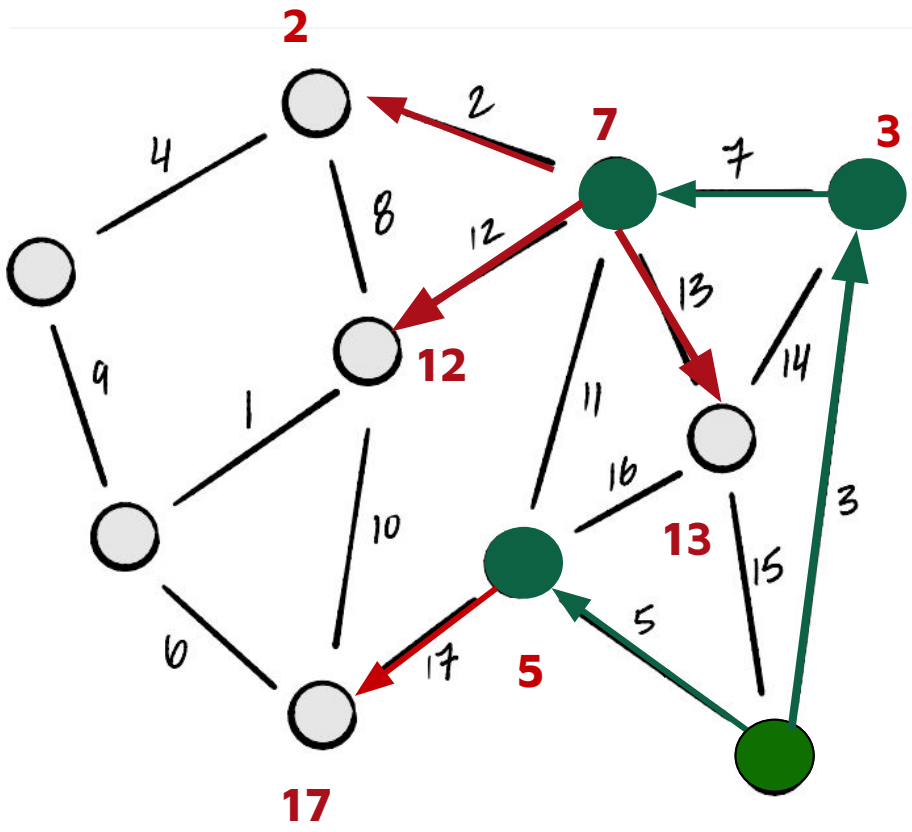

- While $T$ is not a tree:
  - find the node $u \notin T$ with smallest cost.
  - add the edge between $u$ and its estimated "predecessor" to $T$.
  - update estimated cost/pred. of $u$'s neighbors which are not already in the tree.

# Prim's Algorithm, Equivalently




- While $T$ is not a tree:
  - find the node $u \notin T$ with smallest cost.
  - add the edge between $u$ and its estimated "predecessor" to $T$.
  - update estimated cost/pred. of $u$'s neighbors which are not already in the tree.

# Prim's Algorithm, Equivalently




- While $T$ is not a tree:
  - find the node $u \notin T$ with smallest cost.
  - add the edge between $u$ and its estimated "predecessor" to $T$.
  - update estimated cost/pred. of $u$'s neighbors which are not already in the tree.

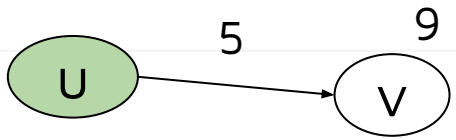
# Recall: Priority Queues

- How do we efficiently find node with smallest cost?
- Priority Queues:
  - `PriorityQueue(priorities)`: creates priority queue from dictionary whose values are priorities.
  - `.extract_min()`: removes and returns key with smallest value.
  - `.decrease_priority(key, value)`: changes key's value.
- We'll use a priority queue to hold nodes not yet added to tree.

```python
def prim(graph, weight):
    tree = UndirectedGraph()

    estimated_predecessor = {node: None for node in graph.nodes}
    cost = {node: float('inf') for node in graph.nodes}
    priority_queue = PriorityQueue(cost)

    while priority_queue:
        u = priority_queue.extract_min()
        if estimated_predecessor[u] is not None:
            tree.add_edge(estimated_predecessor[u], u)
        for v in graph.neighbors(u):
            if weight(u, v) < cost[v] and v not in tree.nodes:
                priority_queue.decrease_priority(v, weight(u, v))
                cost[v] = weight(u, v)
                estimated_predecessor[v] = u
    return tree
```



# Prim and Dijkstra

- This is a lot like Dijkstra's Algorithm for s.p.d.!
- Both: at each step, extract node with smallest cost, update its edges. (**Prim**: only those edges to nodes not in tree).
- **Dijkstra** update of $(u, v)$:

```
cost[v] = min(cost[v], cost[u] + weight(u, v))
```

- **Prim** update of $(u, v)$:

```
cost[v] = min(cost[v], weight(u, v))
```

# Dijkstra is similar to?

```
def dijkstra(graph, weights, source):
    est = {node: float('inf') for node in graph.nodes}
    est[source] = 0
    pred = {node: None for node in graph.nodes}

    priority_queue = PriorityQueue(est)
    while priority_queue:
        u = priority_queue.extract_min()
        for v in graph.neighbors(u):
            changed = update(u, v, weights, est, pred)
            if changed:
                priority_queue.change_priority(v, est[v])

    return est, pred
```

# Time Complexity

# Time Complexity

- A priority queue can be implemented using a heap.
- If a **binary min-heap** is used:
  - `PriorityQueue(est)` takes $\Theta(V)$ time.
  - `.extract_min()` takes $O(\log V)$ time.
  - `.decrease_priority()` takes $O(\log V)$ time.

```
def prim(graph, weight):
    tree = UndirectedGraph()
    estimated_predecessor = {node: None for node in graph.nodes}
    cost = {node: float('inf') for node in graph.nodes}
    priority_queue = PriorityQueue(cost)
    while priority_queue:
        u = priority_queue.extract_min()
        if estimated_predecessor[u] is not None:
            tree.add_edge(estimated_predecessor[u], u)
        for v in graph.neighbors(u):
            if weight(u, v) < cost[v] and v not in tree.nodes:
                priority_queue.decrease_priority(v, weight(u, v))
                cost[v] = weight(u, v)
                estimated_predecessor[v] = u
    return tree
```

$\Theta(V)$

```
def prim(graph, weight):
    tree = UndirectedGraph()
```

Θ(V)

```
    estimated_predecessor = {node: None for node in graph.nodes}
    cost = {node: float('inf') for node in graph.nodes}
    priority_queue = PriorityQueue(cost)

    while priority_queue:
```

**V times**

```
        u = priority_queue.extract_min()
        if estimated_predecessor[u] is not None:
            tree.add_edge(estimated_predecessor[u], u)
        for v in graph.neighbors(u):
            if weight(u, v) < cost[v] and v not in tree.nodes:
                priority_queue.decrease_priority(v, weight(u, v))
                cost[v] = weight(u, v)
                estimated_predecessor[v] = u
    return tree
```

```
def prim(graph, weight):
    tree = UndirectedGraph()
    estimated_predecessor = {node: None for node in graph.nodes}
    cost = {node: float('inf') for node in graph.nodes}
    priority_queue = PriorityQueue(cost)
    while priority_queue:
        u = priority_queue.extract_min()
        if estimated_predecessor[u] is not None:
            tree.add_edge(estimated_predecessor[u], u)
        for v in graph.neighbors(u):
            if weight(u, v) < cost[v] and v not in tree.nodes:
                priority_queue.decrease_priority(v, weight(u, v))
                cost[v] = weight(u, v)
                estimated_predecessor[v] = u
    return tree
```

Θ(V)

**V times**

**log V times to run once**

```
def prim(graph, weight):
    tree = UndirectedGraph()
    estimated_predecessor = {node: None for node in graph.nodes}
    cost = {node: float('inf') for node in graph.nodes}
    priority_queue = PriorityQueue(cost)
    while priority_queue:
        u = priority_queue.extract_min()
        if estimated_predecessor[u] is not None:
            tree.add_edge(estimated_predecessor[u], u)
        for v in graph.neighbors(u):
            if weight(u, v) < cost[v] and v not in tree.nodes:
                priority_queue.decrease_priority(v, weight(u, v))
                cost[v] = weight(u, v)
                estimated_predecessor[v] = u
    return tree
```

Θ(V)

**V times**

**log V times to run once**

**E times**

```
def prim(graph, weight):
    tree = UndirectedGraph()
    estimated_predecessor = {node: None for node in graph.nodes}
    cost = {node: float('inf') for node in graph.nodes}
    priority_queue = PriorityQueue(cost)
    while priority_queue:
        u = priority_queue.extract_min()
        if estimated_predecessor[u] is not None:
            tree.add_edge(estimated_predecessor[u], u)
        for v in graph.neighbors(u):
            if weight(u, v) < cost[v] and v not in tree.nodes:
                priority_queue.decrease_priority(v, weight(u, v))
                cost[v] = weight(u, v)
                estimated_predecessor[v] = u
    return tree
```

Θ(V)

**V times**

**log V times to run once**

**E times**

**log V to run once**

```
def prim(graph, weight):
    tree = UndirectedGraph()
    estimated_predecessor = {node: None for node in graph.nodes}
    cost = {node: float('inf') for node in graph.nodes}
    priority_queue = PriorityQueue(cost)
    while priority_queue:
        u = priority_queue.extract_min()
        if estimated_predecessor[u] is not None:
            tree.add_edge(estimated_predecessor[u], u)
        for v in graph.neighbors(u):
            if weight(u, v) < cost[v] and v not in tree.nodes:
                priority_queue.decrease_priority(v, weight(u, v))
                cost[v] = weight(u, v)
                estimated_predecessor[v] = u
    return tree
```

Θ(V)

**V times**

**log V times to run once**

**E times**

**log V to run once**

O(V + V log V + E log V)

```
def prim(graph, weight):
    tree = UndirectedGraph()
    estimated_predecessor = {node: None for node in graph.nodes}
    cost = {node: float('inf') for node in graph.nodes}
    priority_queue = PriorityQueue(cost)
    while priority_queue:
        u = priority_queue.extract_min()
        if estimated_predecessor[u] is not None:
            tree.add_edge(estimated_predecessor[u], u)
        for v in graph.neighbors(u):
            if weight(u, v) < cost[v] and v not in tree.nodes:
                priority_queue.decrease_priority(v, weight(u, v))
                cost[v] = weight(u, v)
                estimated_predecessor[v] = u
    return tree
```

Θ(V)

**V times**

**log V times to run once**

**E times**

**log V to run once**

O(V log V + E log V)

# Time Complexity

- Using a binary heap...
- Overall: $\Theta(V \log V + E \log V)$.
- Since graph is assumed **connected**, $E = \Omega(V)$.
  - $E \geq V-1$
- So this simplifies to $\Theta(E \log V)$.

# Fibonacci Heaps

- A priority queue can be implemented using a heap.
- If a **Fibonacci** min-heap is used:
  - `PriorityQueue(est)` takes $\Theta(V)$ time.
  - `.extract_min()` takes $\Theta(\log V)$ time.
    - *amortized*
  - **`.decrease_priority()` takes $O(1)$ time.**

```
def prim(graph, weight):
    tree = UndirectedGraph()
    estimated_predecessor = {node: None for node in graph.nodes}
    cost = {node: float('inf') for node in graph.nodes}
    priority_queue = PriorityQueue(cost)
    while priority_queue:
        u = priority_queue.extract_min()
        if estimated_predecessor[u] is not None:
            tree.add_edge(estimated_predecessor[u], u)
        for v in graph.neighbors(u):
            if weight(u, v) < cost[v] and v not in tree.nodes:
                priority_queue.decrease_priority(v, weight(u, v))
                cost[v] = weight(u, v)
                estimated_predecessor[v] = u
    return tree
```

Θ(V)

**V times**

**log V times to run once**

**E times**

**constant to run once**

```
def prim(graph, weight):
    tree = UndirectedGraph()

    estimated_predecessor = {node: None for node in graph.nodes}
    cost = {node: float('inf') for node in graph.nodes}
    priority_queue = PriorityQueue(cost)

    while priority_queue:
        u = priority_queue.extract_min()
        if estimated_predecessor[u] is not None:
            tree.add_edge(estimated_predecessor[u], u)
        for v in graph.neighbors(u):
            if weight(u, v) < cost[v] and v not in tree.nodes:
                priority_queue.decrease_priority(v, weight(u, v))
                cost[v] = weight(u, v)
                estimated_predecessor[v] = u
    return tree
```

Θ(V)

**V times**

**log V times to run once**

**E times**

**constant to run once**

O(V log V + E)

# Time Complexity

- Using a Fibonacci heap...
- Overall: $\Theta(V \log V + E)$.

# Fibonacci vs. Binary Heaps

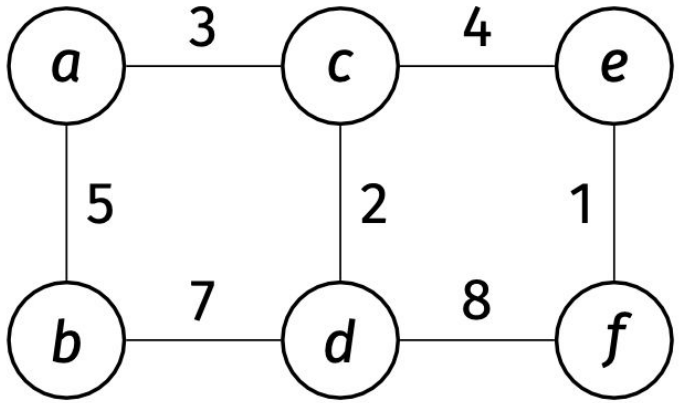
- Using Fibonacci heaps improves time complexity when graph is dense.
- E.g., if $E = \Theta(V^2)$:
  - Prim's with Fibonacci: $\Theta(E) = \Theta(V^2)$
  - Prim's with binary heap: $\Theta(E \log E) = \Theta(V^2 \log V)$. -*slower*
- But Fibonacci heaps are **hard** to implement; have large constants.
- Binary heaps used more in practice despite complexity.

# Correctness of Prim's Algorithm

# Being Greedy

- At every step, we add the lightest edge.
- Is this “safe”?
- **Yes! This is guaranteed to find an MST.**

# Promising Subtrees



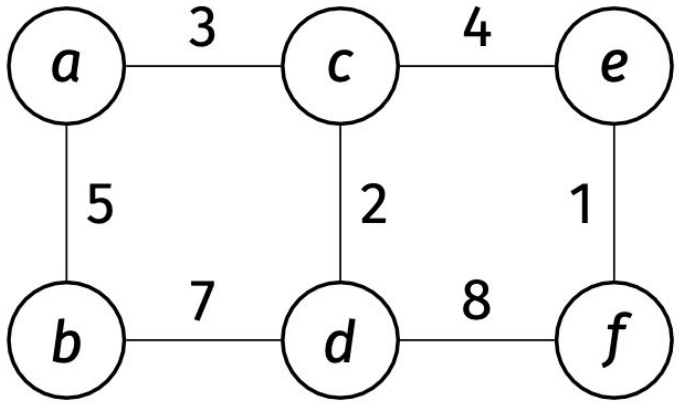

- Let $G = (V, E, \omega)$ be a weighted graph.
- A subgraph $T' = (V', E')$ is **promising** if it is "part" of some MST.
  - That is, it is an "MST in progress"
  - Not necessarily a tree!
- That is, there exists an MST $T = (V, E_{mst})$ such that $E' \subset E_{mst}$.
- **Hint**: a "promising subtree" where $V' = V$ is an MST!

# Main idea

Prim's starts with a promising subtree $T$. At each step, adds lightest edge from a node within $T$ to a node outside of $T$.

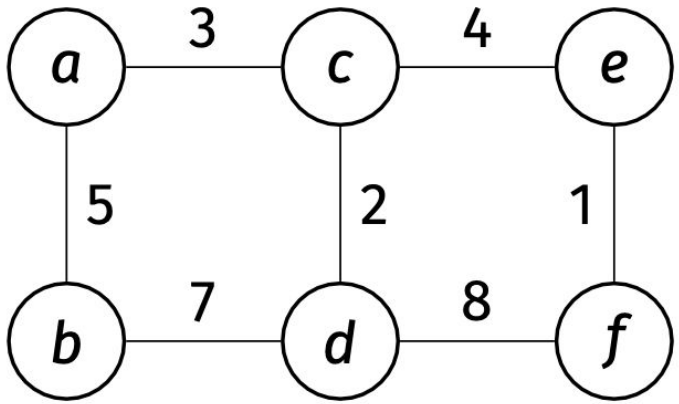
We'll show each new edge results in a larger promising sub-tree. Eventually the promising subtree becomes a full MST.

# Claim




- Let $G = (V, E, \omega)$ be a weighted graph.
- Suppose $T' = (V', E')$ is a promising subtree for an MST of $G$.
- Let $e = (u, v)$ be a **lightest edge** from a node in $T'$ to a node outside of $T'$ (Prim).
- Then adding $(u, v)$ to $T'$ results in another **promising subtree**.

# Proof




- Suppose $T_{mst}$ is an MST that includes $T'$
- If $T_{mst}$ includes $e$, we're done: $T'$ + $e$ is promising.
- If it doesn't include $e$, it must have an edge $f$ that connects $T'$ to rest of the graph.
- Swap $f$ with $e$ in $T_{mst}$. The result is a tree, and it must be a MST since $\omega(e) \leq \omega(f)$.
- So there is an MST that contains $T'$ + $e$.

# Thank you!

**Do you have any questions?**

CampusWire!