# DSC 40B

## Lecture 27 : Minimum Spanning Trees. Kruskal's algorithm

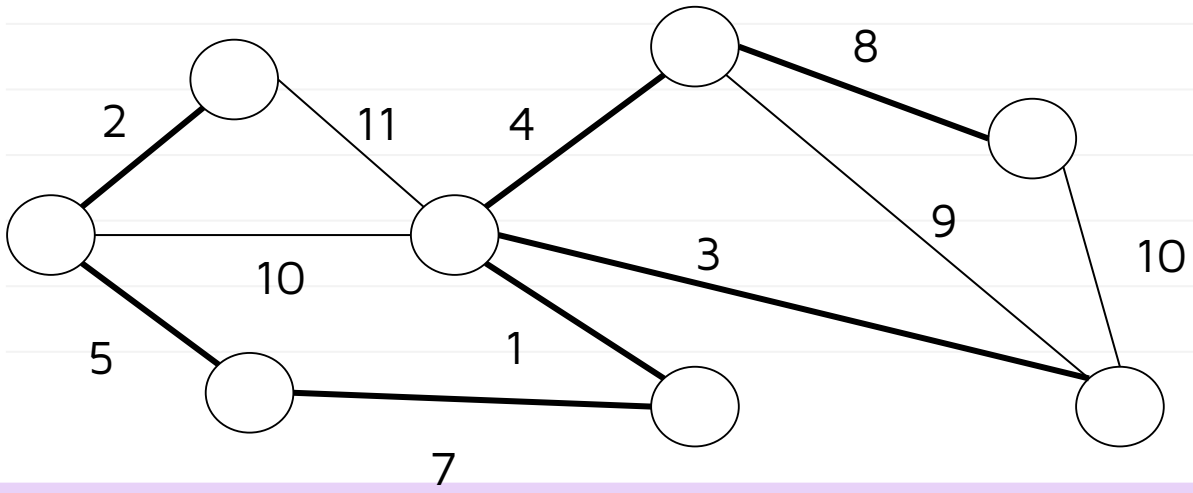
# Midterm 2 details

- Date: December 3rd
- Time: noon
- Location: Our classroom
- Topics:  After midterm1 and up to (including) Bellman-Ford.
  - Lectures 14-25.

# Minimum Spanning Trees

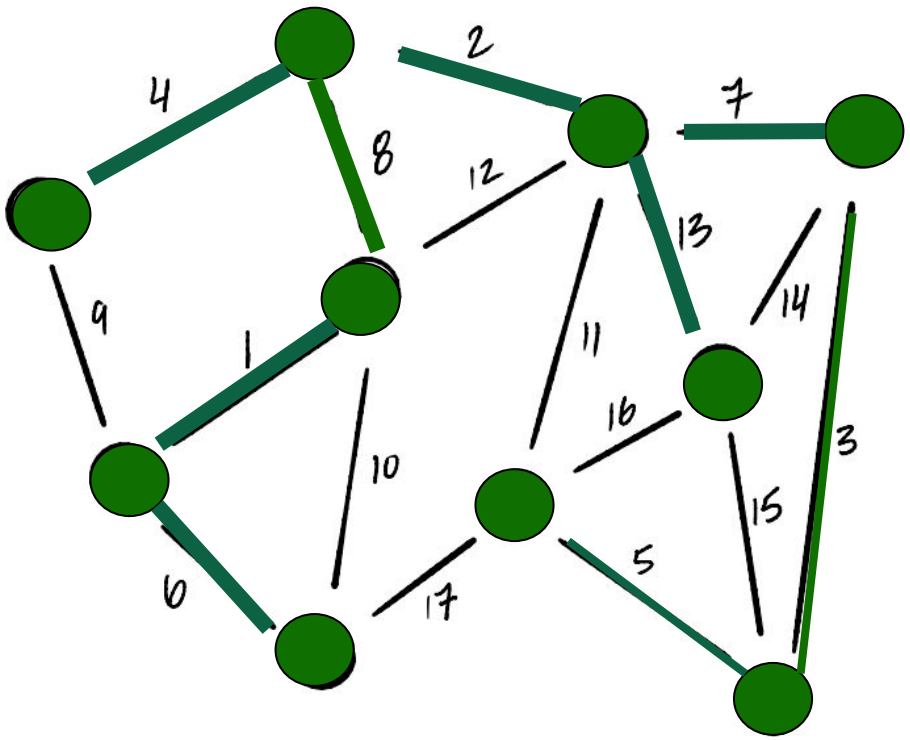
# Last time: Minimum Spanning Tree

- 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

# 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.

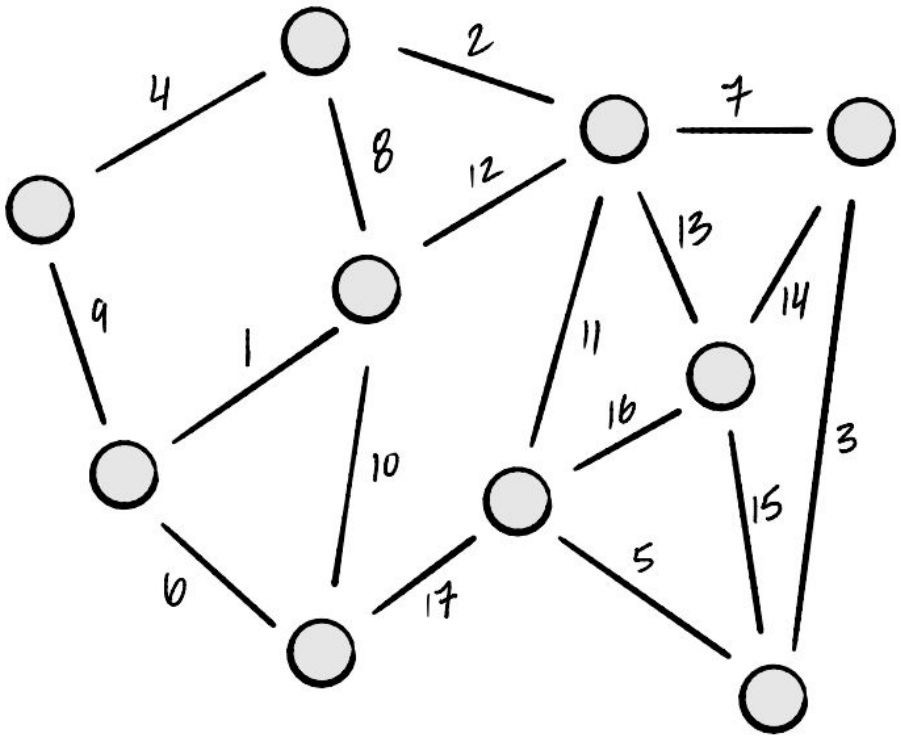
# Last time: Building MSTs

- 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.

# Last time: Two Greedy Approaches

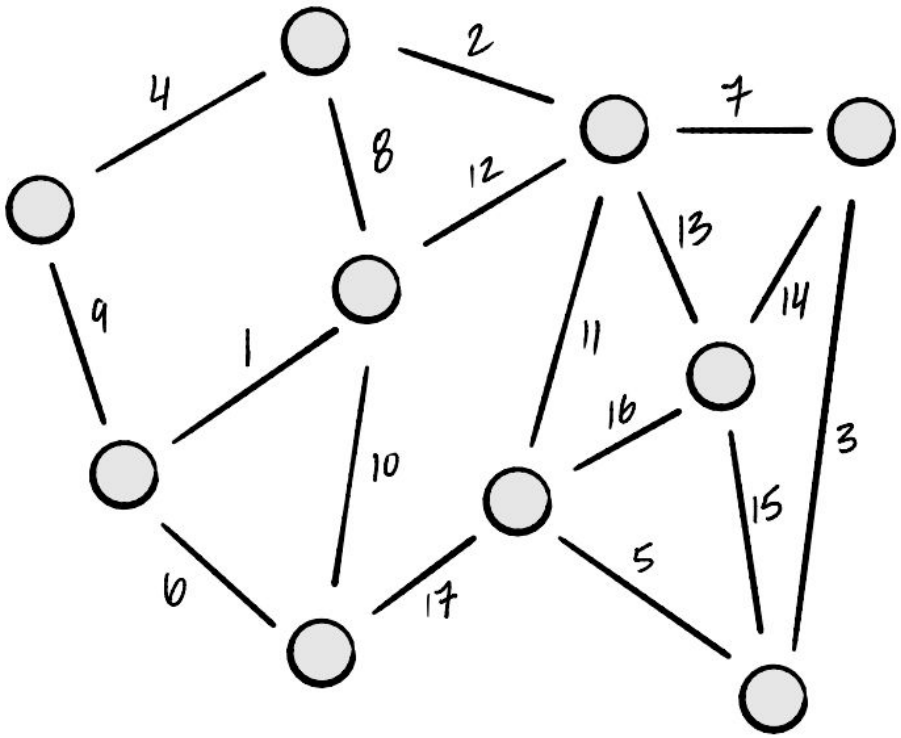
- We'll look at two greedy algorithms:
  - **Last time**: Prim's Algorithm
  - ***Today: Kruskal's Algorithm***
- Differ in the order in which edges are added to tree.
- Also differ in time complexity.

# 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.

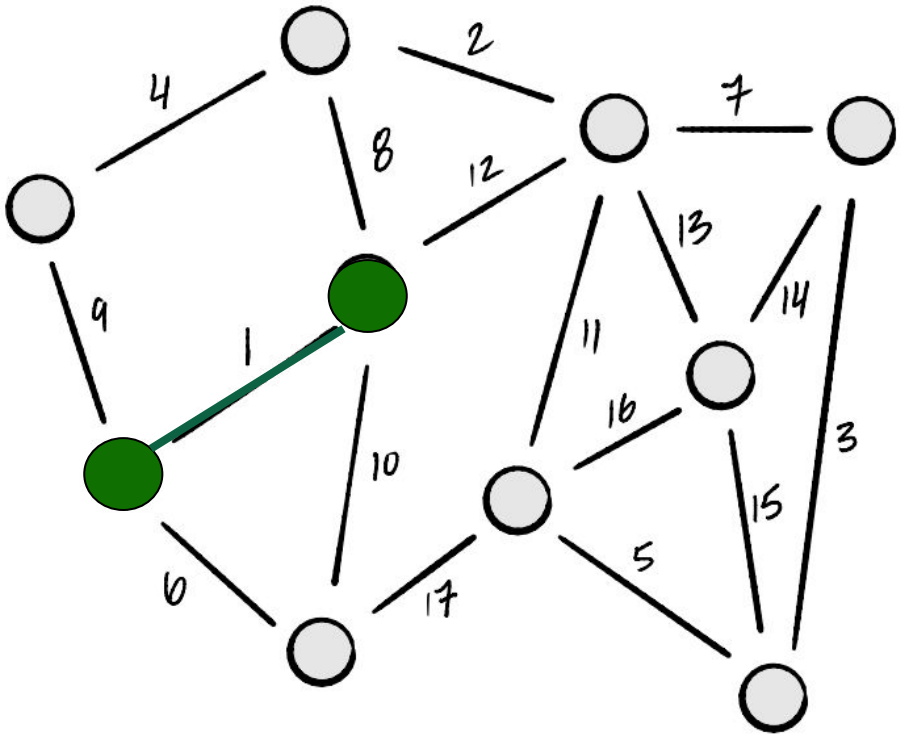
# Kruskal's Algorithm, Informally




- Start with **empty** forest: $T = (V, E_{\text{mst}})$, where $E_{\text{mst}} = \varnothing$.

- Loop through **edges** in *increasing* order of weight.
  - If edge **does not** create a cycle in $T$, add it to $T$.
  - If $T$ is a **spanning tree**, break.

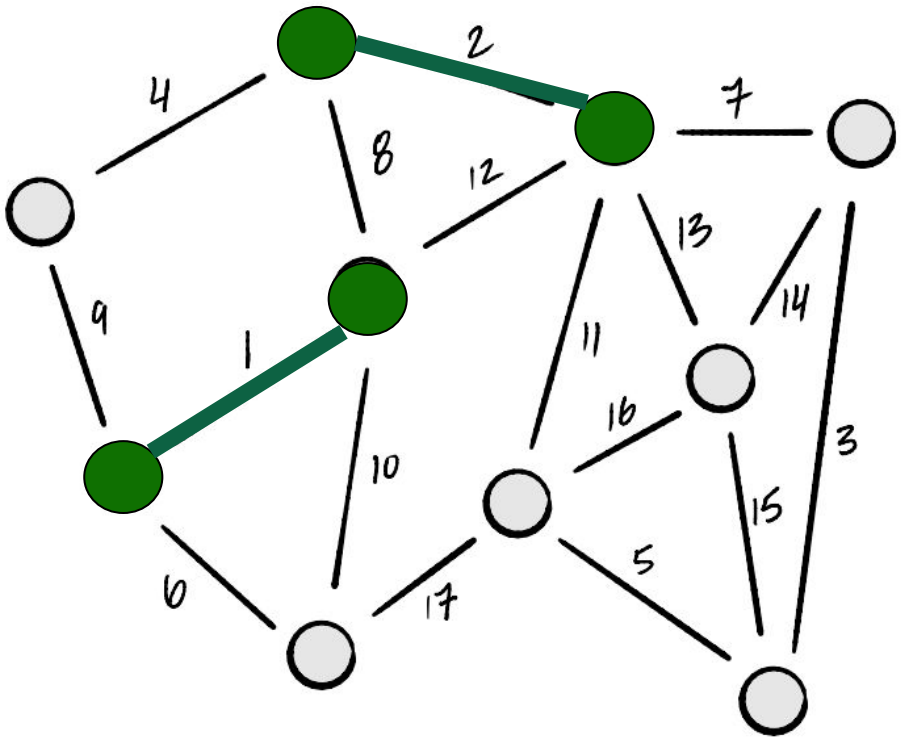
# Kruskal's Algorithm, Informally




- Start with **empty** forest: $T = (V, E_{mst})$, where $E_{mst} = \varnothing$.

- Loop through **edges** in *increasing* order of weight.
  - If edge **does not** create a cycle in $T$, add it to $T$.
  - If $T$ is a **spanning tree**, break.

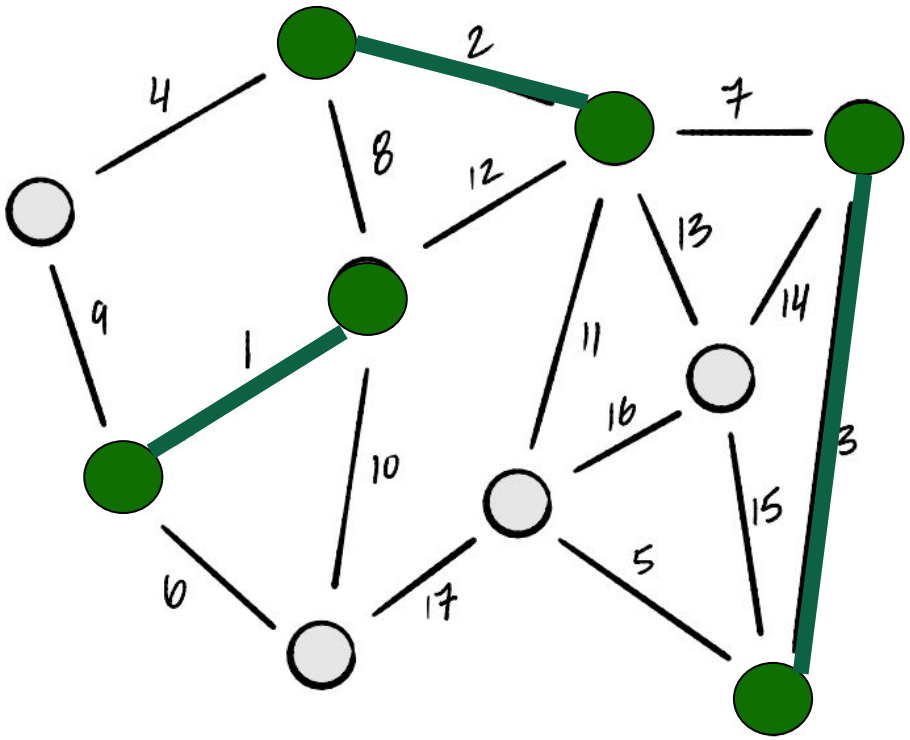
# Kruskal's Algorithm, Informally




- Start with **empty** forest: $T = (V, E_{\text{mst}})$, where $E_{\text{mst}} = \varnothing$.

- Loop through **edges** in *increasing* order of weight.
  - If edge **does not** create a cycle in $T$, add it to $T$.
  - If $T$ is a **spanning tree**, break.

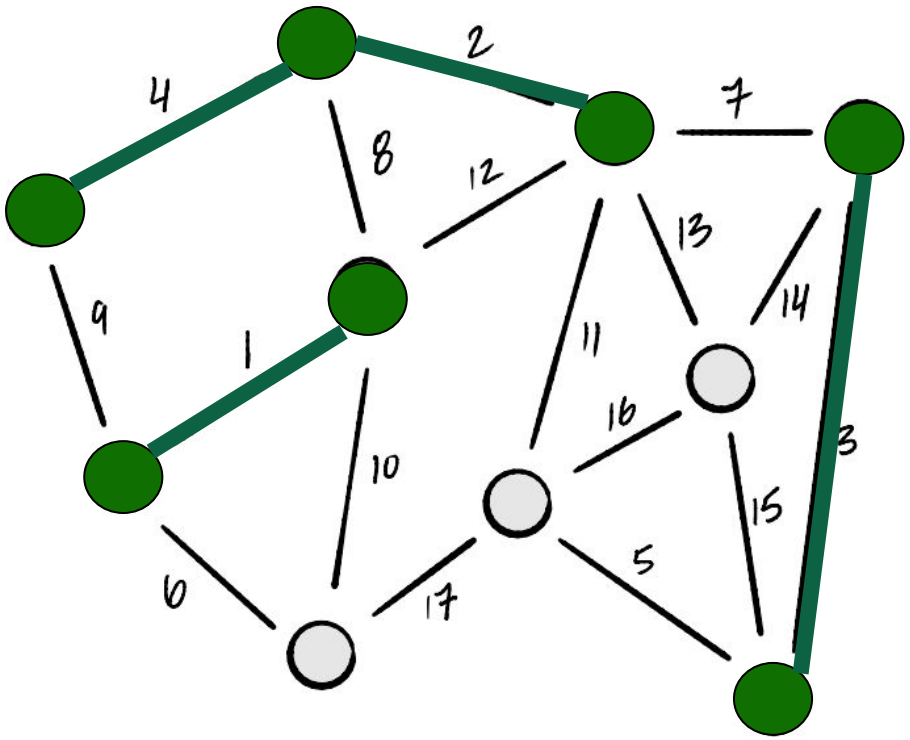
# Kruskal's Algorithm, Informally




- Start with **empty** forest: $T = (V, E_{\text{mst}})$, where $E_{\text{mst}} = \varnothing$.

- Loop through **edges** in *increasing* order of weight.
  - If edge **does not** create a cycle in $T$, add it to $T$.
  - If $T$ is a **spanning tree**, break.

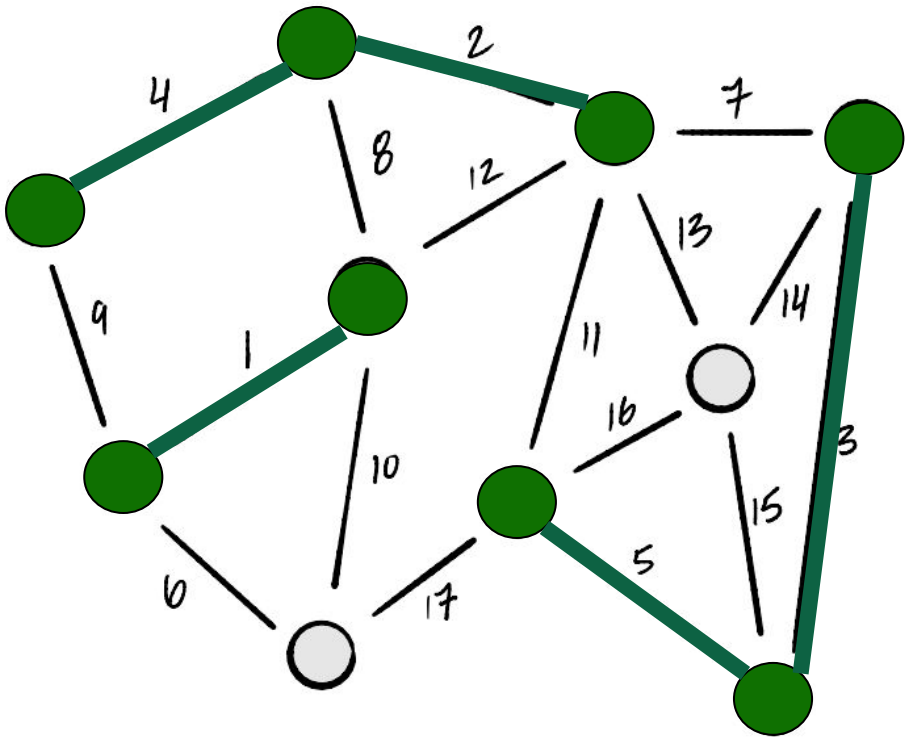
# Kruskal's Algorithm, Informally




- Start with **empty** forest: $T = (V, E_{\text{mst}})$, where $E_{\text{mst}} = \varnothing$.

- Loop through **edges** in *increasing* order of weight.
  - If edge **does not** create a cycle in $T$, add it to $T$.
  - If $T$ is a **spanning tree**, break.

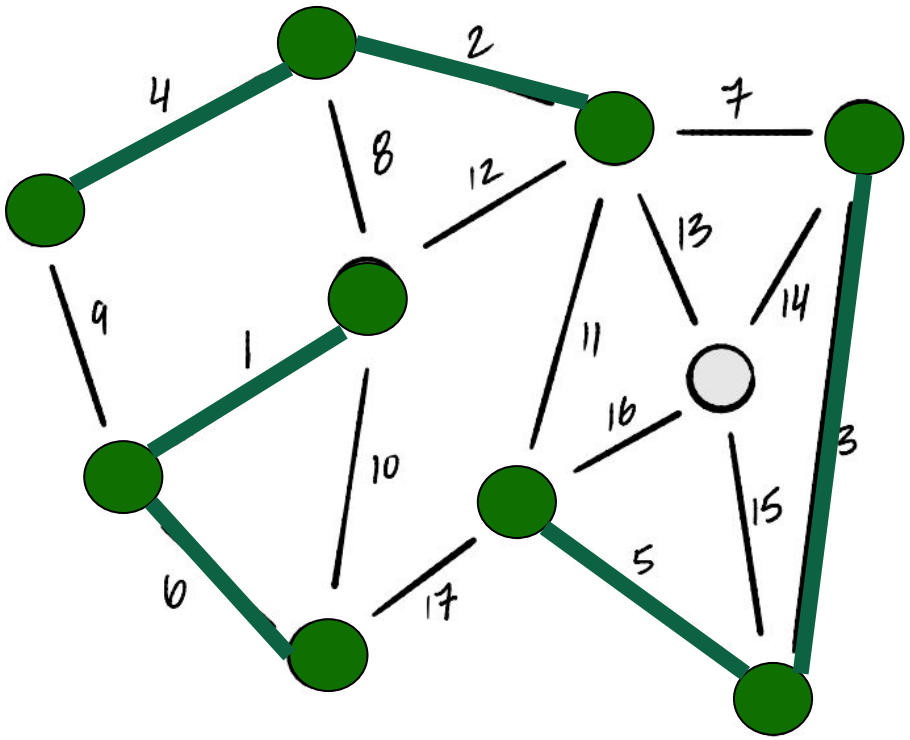
# Kruskal's Algorithm, Informally




- Start with **empty** forest: $T = (V, E_{\text{mst}})$, where $E_{\text{mst}} = \varnothing$.

- Loop through **edges** in *increasing* order of weight.
  - If edge **does not** create a cycle in $T$, add it to $T$.
  - If $T$ is a **spanning tree**, break.

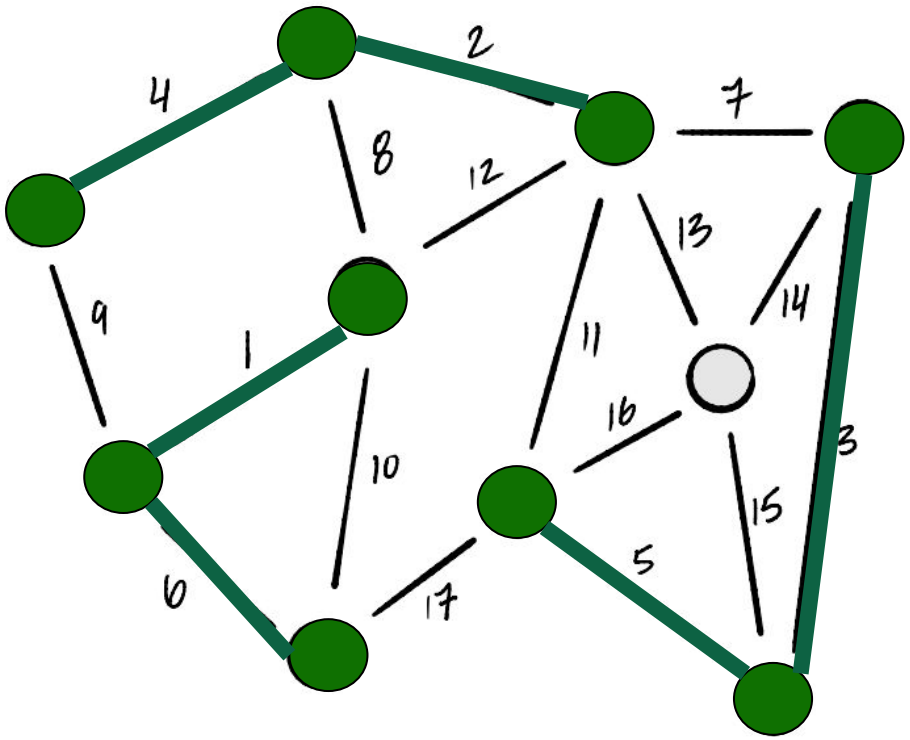
# Kruskal's Algorithm, Informally




- Start with **empty** forest: $T = (V, E_{\text{mst}})$, where $E_{\text{mst}} = \varnothing$.

- Loop through **edges** in *increasing* order of weight.
  - If edge **does not** create a cycle in $T$, add it to $T$.
  - If $T$ is a **spanning tree**, break.

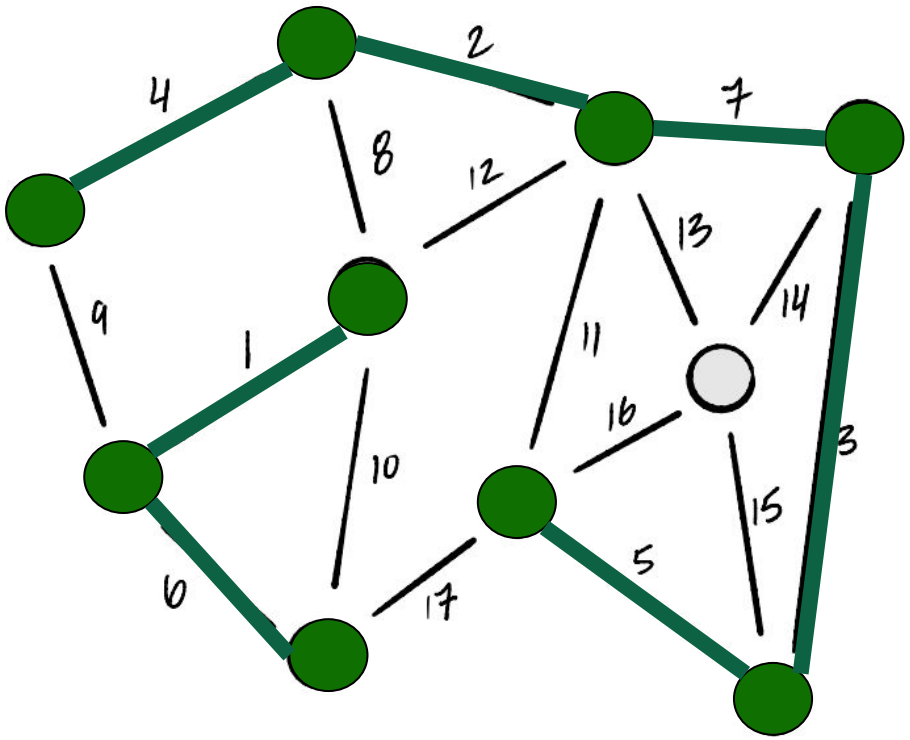
# Kruskal's Algorithm, Informally




- Start with **empty** forest: $T = (V, E_{\text{mst}})$, where $E_{\text{mst}} = \varnothing$.

- Loop through **edges** in *increasing* order of weight.
  - If edge **does not** create a cycle in $T$, add it to $T$.
  - If $T$ is a **spanning tree**, break.

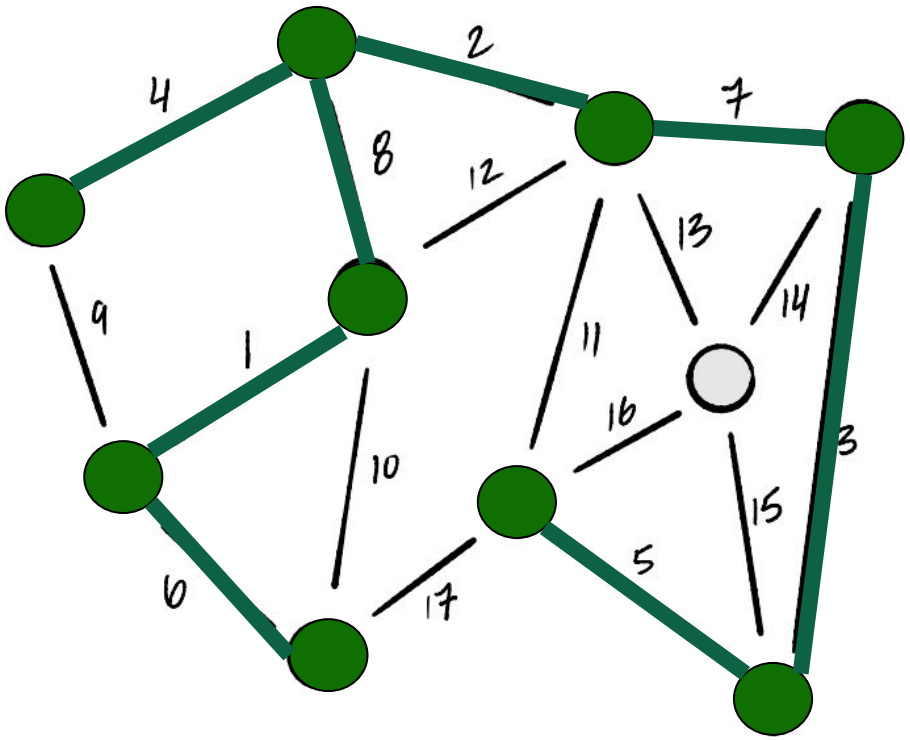
# Kruskal's Algorithm, Informally




- Start with **empty** forest: $T = (V, E_{\text{mst}})$, where $E_{\text{mst}} = \varnothing$.

- Loop through **edges** in *increasing* order of weight.
  - If edge **does not** create a cycle in $T$, add it to $T$.
  - If $T$ is a **spanning tree**, break.

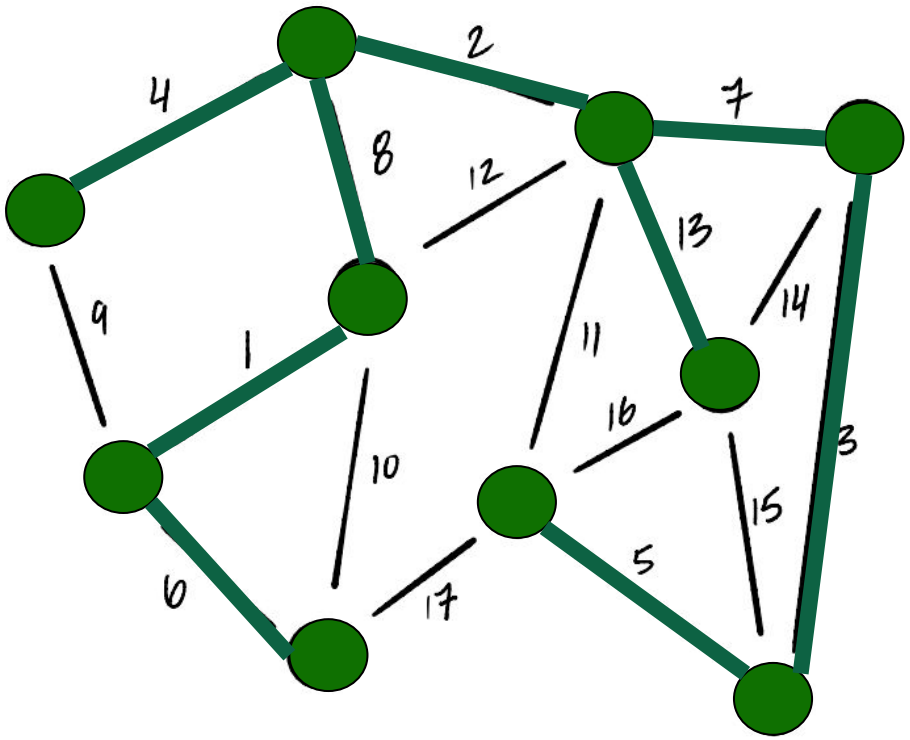
# Kruskal's Algorithm, Informally




- Start with **empty** forest: $T = (V, E_{\text{mst}})$, where $E_{\text{mst}} = \varnothing$.

- Loop through **edges** in *increasing* order of weight.
  - If edge **does not** create a cycle in $T$, add it to $T$.
  - If $T$ is a **spanning tree**, break.

# Kruskal's Algorithm, Informally




- Start with **empty** forest: $T = (V, E_{\text{mst}})$, where $E_{\text{mst}} = \varnothing$.

- Loop through **edges** in *increasing* order of weight.
  - If edge **does not** create a cycle in $T$, add it to $T$.
  - If $T$ is a **spanning tree**, break.

# Being Greedy

- **Prim**: add the node with smallest estimated cost and update neighbors.
  - Works locally, “grows” a connected tree.
- **Kruskal**: add the edge with smallest weight.
  - As long as it doesn’t make a cycle.
  - Edge can be anywhere in graph.

# Kruskal's Algorithm (Pseudocode)

```
def kruskal(graph, weights):
    mst = UndirectedGraph()

    # sort edges in ascending order by weight
    sorted_edges = sorted(graph.edges, key=weights.get)

    for (u, v) in sorted_edges:
    # if u and v are not already connected
    if ...:
        mst.add_edge(u, v)

        # (optional) if mst is now a spanning tree, break
        if len(mst.edges) == len(graph.nodes) - 1:
            break

    return mst
```

# Checking for Connectivity

- Each iteration: check if $u$ and $v$ are connected in $T = (V, E_{mst})$.

- We *could* do a DFS/BFS on each iteration…
  - $\Theta(V + E_{mst}) = \Theta(V)$ each time.
  - **Expensive!**

- **Remember:**
  - If you're computing something **once**, use a *fast algorithm*.
  - If you're computing it **repeatedly**, consider a *data structure*.

# Disjoint Set Forests

- Represent a collection of disjoint sets.

{{1, 5, 6}, {2, 3}, {0}, {4}}

- `.union(x, y)`: Union the sets containing $x$ and $y$.
- `.in_same_set(x, y)`: Return **True/False** if $x$ and $y$ are in the same set*.

**Usually implemented as a `.find(x)` method returning representative of set containing x.*

# Example

```
>>> # create a DSF with {{0}, {1}, {2}, {3}, {4}, {5}}
>>> dsf = DisjointSetForest([0, 1, 2, 3, 4, 5])
>>> dsf.union(0, 3)
>>> dsf.union(1, 4)
>>> dsf.union(3, 1)
>>> dsf.union(2, 5)
>>> # dsf now represents {{0, 1, 3, 4}, {2, 5}}
>>> dsf.in_same_set(0, 3)
True
>>> dsf.in_same_set(0, 2)
False
```

# Disjoint Set Forests

- Operations take $\Theta(\alpha(n))$ time, where $n$ is number of objects in collection.
- $\alpha(n)$ is the **inverse Ackermann function.**
- It grows **very**, **very** slowly.
- *Essentially* constant time.

# Disjoint Set Forests

- Can be used to keep track of Connected Components of a **dynamic graph.**
- Nodes of Connected Components are disjoint sets.
  - Add an edge $(u, v)$: `.union(u, v)`
  - Check if $u$ and $v$ are connected: `.in_same_set(u, v)`
- To check if $u$, $v$ are already connected:
  - BFS/DFS: $\Theta(V)$ each time.
  - DSF: $\Theta(\alpha(V))$ each time (essentially $\Theta(1)$).

```python
def kruskal(graph, weights):
    mst = UndirectedGraph()

    # place each node in its own disjoint set
    components = DisjointSetForest(graph.nodes)

    # sort edges in ascending order by weight
    sorted_edges = sorted(graph.edges, key=weights)

    for (u, v) in sorted_edges:
        if not components.in_same_set(u, v):
            mst.add_edge(u, v)
            components.union(u, v)

            # (optional) if mst is now a spanning tree, break
            if len(mst.edges) == len(graph.nodes) - 1:
                break

    return mst
```

## Time Complexity

$\Theta(V)$

```python
def kruskal(graph, weights):
    mst = UndirectedGraph()

    # place each node in its own disjoint set
    components = DisjointSetForest(graph.nodes)

    # sort edges in ascending order by weight
    sorted_edges = sorted(graph.edges, key=weights)

    for (u, v) in sorted_edges:
        if not components.in_same_set(u, v):
            mst.add_edge(u, v)
            components.union(u, v)

            # (optional) if mst is now a spanning tree, break
            if len(mst.edges) == len(graph.nodes) - 1:
                break

    return mst
```

## Time Complexity

Θ(V)

Θ(E log E)

```
def kruskal(graph, weights):
    mst = UndirectedGraph()

    # place each node in its own disjoint set
    components = DisjointSetForest(graph.nodes)

    # sort edges in ascending order by weight
    sorted_edges = sorted(graph.edges, key=weights)

    for (u, v) in sorted_edges:
        if not components.in_same_set(u, v):
            mst.add_edge(u, v)
            components.union(u, v)

            # (optional) if mst is now a spanning tree, break
            if len(mst.edges) == len(graph.nodes) - 1:
                break

    return mst
```

## Time Complexity

Θ(V)

Θ(E log E)

O(E )

```
def kruskal(graph, weights):
    mst = UndirectedGraph()

    # place each node in its own disjoint set
    components = DisjointSetForest(graph.nodes)

    # sort edges in ascending order by weight
    sorted_edges = sorted(graph.edges, key=weights)

    for (u, v) in sorted_edges:
        if not components.in_same_set(u, v):
            mst.add_edge(u, v)
            components.union(u, v)

            # (optional) if mst is now a spanning tree, break
            if len(mst.edges) == len(graph.nodes) - 1:
                break

    return mst
```

## Time Complexity

Θ(V)

Θ(E log E)

O(E )

$\alpha$(V) per line

```
def kruskal(graph, weights):
    mst = UndirectedGraph()

    # place each node in its own disjoint set
    components = DisjointSetForest(graph.nodes)

    # sort edges in ascending order by weight
    sorted_edges = sorted(graph.edges, key=weights)

    for (u, v) in sorted_edges:
        if not components.in_same_set(u, v):
            mst.add_edge(u, v)
            components.union(u, v)

            # (optional) if mst is now a spanning tree, break
            if len(mst.edges) == len(graph.nodes) - 1:
                break

    return mst
```

## Time Complexity

Θ(V)

Θ(E log E)

O(E )

$\alpha$(V) per line

O(E $\alpha$(V) ) for loop

```
def kruskal(graph, weights):
    mst = UndirectedGraph()

    # place each node in its own disjoint set
    components = DisjointSetForest(graph.nodes)

    # sort edges in ascending order by weight
    sorted_edges = sorted(graph.edges, key=weights)

    for (u, v) in sorted_edges:
        if not components.in_same_set(u, v):
            mst.add_edge(u, v)
            components.union(u, v)

    # (optional) if mst is now a spanning tree, break
    if len(mst.edges) == len(graph.nodes) - 1:
        break

return mst
```

## Time Complexity

```
def kruskal(graph, weights):
    mst = UndirectedGraph()

    # place each node in its own disjoint set
    components = DisjointSetForest(graph.nodes)

    # sort edges in ascending order by weight
    sorted_edges = sorted(graph.edges, key=weights)

    for (u, v) in sorted_edges:
        if not components.in_same_set(u, v):
            mst.add_edge(u, v)
            components.union(u, v)

            # (optional) if mst is now a spanning tree, break
            if len(mst.edges) == len(graph.nodes) - 1:
                break

    return mst
```

Θ(V)

Θ(E log E)

O(E )

$\alpha$(V) per line

O(E $\alpha$(V) ) for loop

Θ(V + E log E + E $\alpha$(V))

## Time Complexity

```
def kruskal(graph, weights):
    mst = UndirectedGraph()

    # place each node in its own disjoint set
    components = DisjointSetForest(graph.nodes)

    # sort edges in ascending order by weight
    sorted_edges = sorted(graph.edges, key=weights)

    for (u, v) in sorted_edges:
        if not components.in_same_set(u, v):
            mst.add_edge(u, v)
            components.union(u, v)

            # (optional) if mst is now a spanning tree, break
            if len(mst.edges) == len(graph.nodes) - 1:
                break

    return mst
```

Θ(V)

Θ(E log E)

O(E )

$\alpha$(V) per line

O(E $\alpha$(V) ) for loop

Θ(V + E log E)

## Time Complexity

```
def kruskal(graph, weights):
    mst = UndirectedGraph()

    # place each node in its own disjoint set
    components = DisjointSetForest(graph.nodes)

    # sort edges in ascending order by weight
    sorted_edges = sorted(graph.edges, key=weights)

    for (u, v) in sorted_edges:
        if not components.in_same_set(u, v):
            mst.add_edge(u, v)
            components.union(u, v)

        # (optional) if mst is now a spanning tree, break
        if len(mst.edges) == len(graph.nodes) - 1:
            break

    return mst
```

Θ(V)

Θ(E log E)

O(E )

$\alpha$(V) per line

O(E $\alpha$(V) ) for loop

Θ(E log E) if assume connected

# Time Complexity

- Assume graph is connected. Then $E = \Omega(V)$.
- Kruskal's takes $\Theta(E \log E) = \Theta(E \log V)$ time.
  - Dominated by sorting the edges.
- **Note**: if graph disconnected, Kruskal's produces a **minimum spanning forest**.

# Kruskal vs. Prim

# Kruskal v. Prim

- Both algorithms for computing MSTs.
- Which is “better”?
- There’s no clear winner.

# Time Complexity

- **Prim**:
  - Binary heap: $\Theta(V \log V + E \log V)$
  - Fibonacci heap: $\Theta(V \log V + E)$
- **Kruskal**: $\Theta(E \log V)$
- If the graph is dense, $E = \Theta(V^2)$, and Prim's with Fibonacci heap "wins".
  - $\Theta(V^2)$ versus $\Theta(V^2 \log V)$.

# Not so fast...

- Fibonacci heaps are hard to implement, **high** overhead.
- Prim's will be faster for **very large dense graphs**.
- But Kruskal's may be faster for **smaller dense graphs**.
- The right choice depends on your application.

# Main Idea

Asymptotic time complexity isn’t everything. For small inputs, the “inefficient” algorithm may beat the “efficient” one. There’s also ease of implementation to consider.

# MSTs and Clustering Next Time :)

# Thank you!

**Do you have any questions?**

CampusWire!