Definition: Tree

A connected graph with no cycles

Note: A tree does not need a root, but they often do!
Definition: Tree

A connected graph with no cycles

Pick some arbitrary root node and rearrange tree
Definition: Spanning Tree

A Tree \( T = (V_T, E_T) \) which connects (“spans”) all the nodes in a graph \( G = (V, E) \)

How many edges does \( T \) have?
\( V - 1 \)

Any set of \( V-1 \) edges in the graph that doesn’t have any cycles is guaranteed to be a spanning tree!

Pick some arbitrary root node and rearrange tree

Any set of \( V-1 \) edges that connects all the nodes in the graph is guaranteed to be a spanning tree!
Definition: Minimum Spanning Tree

A Tree \( T = (V_T, E_T) \) which connects ("spans") all the nodes in a graph \( G = (V, E) \), that has minimal cost

\[
\text{Cost}(T) = \sum_{e \in E_T} w(e)
\]
Kruskal’s Algorithm

Start with an empty tree $A$
Add to $A$ the lowest-weight edge that does not create a cycle
Kruskal’s Algorithm

Start with an empty tree $A$
Add to $A$ the lowest-weight edge that does not create a cycle
Kruskal’s Algorithm

Start with an empty tree $A$
Add to $A$ the lowest-weight edge that does not create a cycle
Kruskal’s Algorithm

Start with an empty tree \( A \)
Add to \( A \) the lowest-weight edge that does not create a cycle
Kruskal’s Algorithm

Start with an empty tree $A$

Add to $A$ the lowest-weight edge that does not create a cycle
Kruskal’s Algorithm

Start with an empty tree $A$

Add to $A$ the lowest-weight edge that does not create a cycle
Definition: Cut

A Cut of graph $G = (V, E)$ is a partition of the nodes into two sets, $S$ and $V - S$.

Edge $(v_1, v_2) \in E$ crosses a cut if $v_1 \in S$ and $v_2 \in V - S$ (or opposite), e.g. $(A, C)$.

A set of edges $R$ Respects a cut if no edges cross the cut, e.g. $R = \{(A, B), (E, G), (F, G)\}$.
Cut Theorem

If a set of edges $A$ is a subset of a minimum spanning tree $T$, let $(S, V - S)$ be any cut which $A$ respects. Let $e$ be the least-weight edge which crosses $(S, V - S)$. $A \cup \{e\}$ is also a subset of a minimum spanning tree.
Cut Theorem

If a set of edges $A$ is a subset of a minimum spanning tree $T$, let $(S, V - S)$ be any cut which $A$ respects. Let $e$ be the least-weight edge which crosses $(S, V - S)$. $A \cup \{e\}$ is also a subset of a minimum spanning tree.
Cut Theorem

If a set of edges $A$ is a subset of a minimum spanning tree $T$, let $(S, V - S)$ be any cut which $A$ respects. Let $e$ be the least-weight edge which crosses $(S, V - S)$. $A \cup \{e\}$ is also a subset of a minimum spanning tree.
Cut Theorem

If a set of edges $A$ is a subset of a minimum spanning tree $T$, let $(S, V - S)$ be any cut which $A$ respects. Let $e$ be the least-weight edge which crosses $(S, V - S)$. $A \cup \{e\}$ is also a subset of a minimum spanning tree.
Cut Theorem

If a set of edges $A$ is a subset of a minimum spanning tree $T$, let $(S, V - S)$ be any cut which $A$ respects. Let $e$ be the least-weight edge which crosses $(S, V - S)$. $A \cup \{e\}$ is also a subset of a minimum spanning tree.
Proof of Kruskal’s Algorithm

Start with an empty tree $A$
Repeat $V - 1$ times:
    Add the min-weight edge that doesn’t cause a cycle

Proof: Suppose we have some arbitrary set of edges $A$ that Kruskal’s has already selected to include in the MST. $e = (F, G)$ is the edge Kruskal’s selects to add next

We know that there cannot exist a path from $F$ to $G$ using only edges in $A$ because $e$ does not cause a cycle

We can cut the graph therefore into 2 disjoint sets:
- nodes reachable from $G$ using edges in $A$
- All other nodes

$e$ is the minimum cost edge that crosses this cut, so by the Cut Theorem, Kruskal’s is optimal!
Kruskal’s Algorithm Runtime

Start with an empty tree $A$

Repeat $V - 1$ times:

Add the min-weight edge that doesn’t cause a cycle

Keep edges in a Disjoint-set data structure (very fancy)

$O(E \log V)$
General MST Algorithm

Start with an empty tree $A$

Repeat $V - 1$ times:

Pick a cut $(S, V - S)$ which $A$ respects (typically implicitly)

Add the min-weight edge which crosses $(S, V - S)$
Prim’s Algorithm

Start with an empty tree $A$

Repeat $V - 1$ times:

Pick a cut $(S, V - S)$ which $A$ respects

Add the min-weight edge which crosses $(S, V - S)$

$S$ is all endpoint of edges in $A$

$e$ is the min-weight edge that grows the tree
Prim’s Algorithm

Start with an empty tree $A$

Pick a start node

Repeat $V - 1$ times:

Add the min-weight edge which connects to node in $A$ with a node not in $A$
Prim’s Algorithm

Start with an empty tree $A$

Pick a start node

Repeat $V - 1$ times:

Add the min-weight edge which connects to node in $A$ with a node not in $A$
Prim’s Algorithm

Start with an empty tree $A$

Pick a start node

Repeat $V - 1$ times:

Add the min-weight edge which connects to node in $A$ with a node not in $A$
Prim’s Algorithm

Start with an empty tree $A$

Pick a start node

Repeat $V - 1$ times:

Add the min-weight edge which connects to node in $A$ with a node not in $A$
Prim’s Algorithm

Start with an empty tree $A$

Pick a start node

Repeat $V - 1$ times:

Add the min-weight edge which connects to node in $A$ with a node not in $A$

Keep edges in a Heap

$O(E \log V)$
Dijkstra’s Algorithm

```java
int dijkstras(graph, start, end){
    distances = [∞, ∞, ∞,...]; // one index per node
    done = [False,False,False,...]; // one index per node
    PQ = new minheap();
    PQ.insert(0, start); // priority=0, value=start
    distances[start] = 0;
    while (!PQ.isEmpty){
        current = PQ.extract();
        done[current] = true;
        for (neighbor : current.neighbors){
            if (!done[neighbor]){
                new_dist = distances[current]+weight(current,neighbor);
                if(distances[neighbor] == ∞){
                    distances[neighbor] = new_dist;
                    PQ.insert(new_dist, neighbor);
                }
                if (new_dist < distances[neighbor]){  
                    distances[neighbor] = new_dist;
                    PQ.decreaseKey(new_dist, neighbor); }
            }
        }
    }
    return distances[end]
}
```
Prims's Algorithm

```java
int primss(graph, start, end){
    distances = [\infty, \infty, \infty,...]; // one index per node
    done = [False,False,False,...]; // one index per node
    PQ = new minheap();
    PQ.insert(0, start); // priority=0, value=start
    distances[start] = 0;
    while (!PQ.isEmpty){
        current = PQ.extract();
        done[current] = true;
        for (neighbor : current.neighbors){
            if (!done[neighbor]){  
                new_dist = weight(current,neighbor);
                if(distances[neighbor] == \infty){
                    distances[neighbor] = new_dist;
                    PQ.insert(new_dist, neighbor);
                }
                else if (new_dist < distances[neighbor]){
                    distances[neighbor] = new_dist;
                    PQ.decreaseKey(new_dist, neighbor);
                }
            }
        }
    }
    return distances[end]
}
```
Dijkstra’s Algorithm

```java
int dijkstras(graph, start, end){
    distances = [∞, ∞, ∞,...]; // one index per node
    done = [False, False, False,...]; // one index per node
    PQ = new minheap();
    PQ.insert(0, start); // priority=0, value=start
    distances[start] = 0;
    while (!PQ.isEmpty){
        current = PQ.extract();
        done[current] = true;
        for (neighbor : current.neighbors){
            if (!done[neighbor]){
                new_dist = distances[current]+weight(current,neighbor);
                if(distances[neighbor] == ∞){
                    distances[neighbor] = new_dist;
                    PQ.insert(new_dist, neighbor);
                }
                if (new_dist < distances[neighbor]){  
                    distances[neighbor] = new_dist;
                    PQ.decreaseKey(new_dist, neighbor);
                }
            }
        }
    }
    return distances[end]
}
```
int primss(graph, start, end) {
    distances = [∞, ∞, ∞, ...]; // one index per node
    done = [False, False, False, ...]; // one index per node
    PQ = new minheap();
    PQ.insert(0, start); // priority=0, value=start
    distances[start] = 0;
    while (!PQ.isEmpty) {
        current = PQ.extract();
        done[current] = true;
        for (neighbor : current.neighbors) {
            if (!done[neighbor]) {
                new_dist = weight(current, neighbor);
                if (distances[neighbor] == ∞) {
                    distances[neighbor] = new_dist;
                    PQ.insert(new_dist, neighbor);
                }
                if (new_dist < distances[neighbor]) {
                    distances[neighbor] = new_dist;
                    PQ.decreaseKey(new_dist, neighbor);
                }
            }
        }
    }
    return distances[end]
}

Prims’s Algorithm