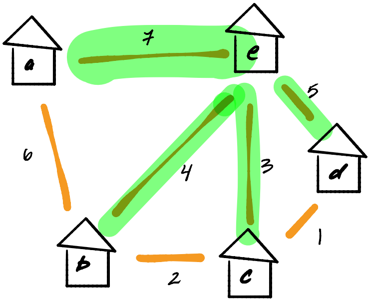
DSC 40B

Theoretical Foundations II

Lecture 16 | Part 1

Minimum Spanning Trees

Today's Problem



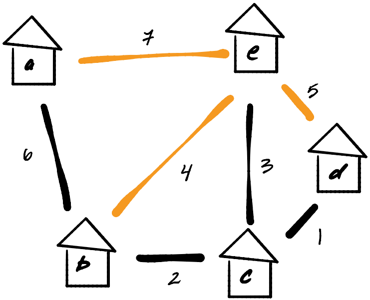
$7+4+3+5$

$= 19$

- Choose a set of dirt roads to pave so that:
  - can get between any two buildings only on paved roads;
  - total cost is minimized.

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

Today's Problem



- Choose a set of dirt roads to pave so that:
  - can get between any two buildings only on paved roads;
  - total cost is minimized.

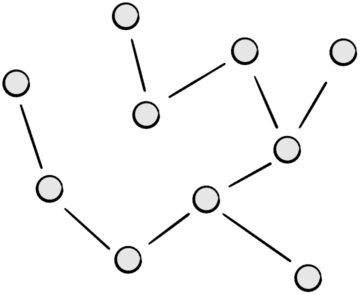
- Solution: compute a **minimum spanning tree**.

Trees

An undirected graph $T = (V, E)$ is a **tree** if

- it is connected; and
- it is acyclic.

Example: a **tree**.

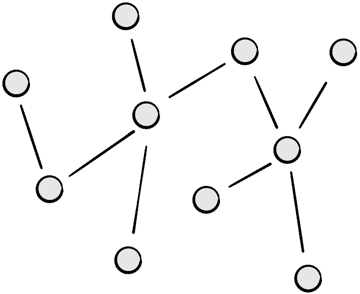


Trees

An undirected graph $T = (V, E)$ is a **tree** if

- it is connected; and
- it is acyclic.

Example: a **tree**.

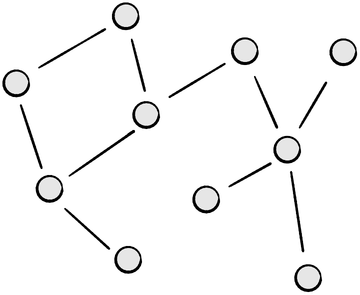


Trees

An undirected graph $T = (V, E)$ is a **tree** if

- it is connected; and
- it is acyclic.

Example: **not** a tree.

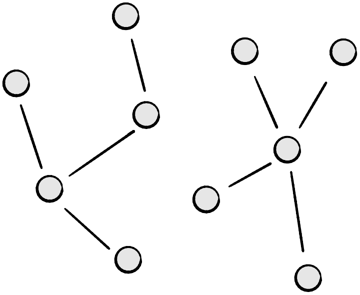


Trees

An undirected graph $T = (V, E)$ is a **tree** if

- it is connected; and
- it is acyclic.

Example: **not** a tree.

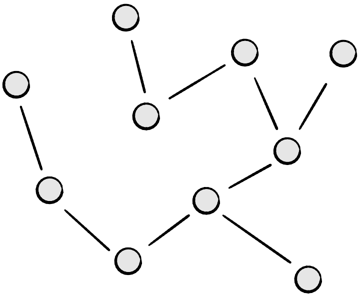


Trees: Equivalent Definition

An undirected graph $T = (V, E)$ is a **tree** if

- it is connected; and
- $|E| = |V| - 1$.

Example: a **tree**.

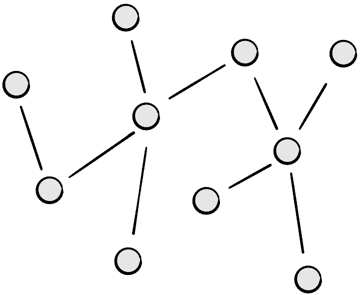


Trees: Equivalent Definition

An undirected graph $T = (V, E)$ is a **tree** if

- it is connected; and
- $|E| = |V| - 1$.

Example: a **tree**.

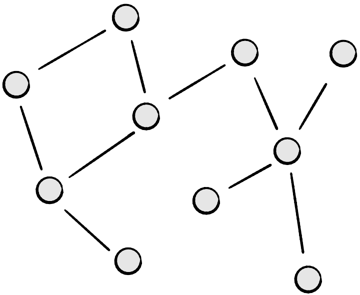


Trees: Equivalent Definition

An undirected graph $T = (V, E)$ is a **tree** if

- it is connected; and
- $|E| = |V| - 1$.

Example: **not** a tree.

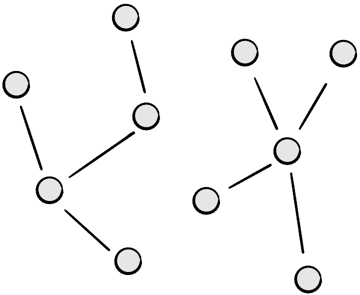


Trees: Equivalent Definition

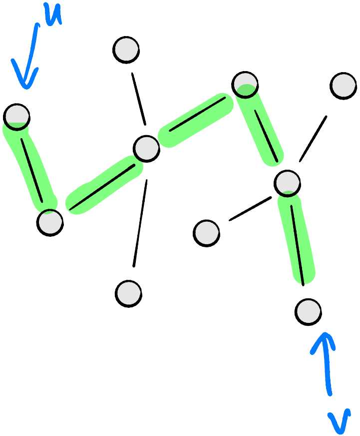
An undirected graph $T = (V, E)$ is a **tree** if

- it is connected; and
- $|E| = |V| - 1$.

Example: **not** a tree.



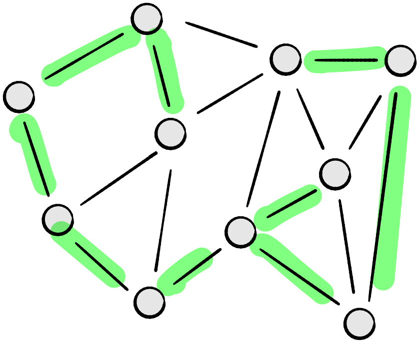
Tree Properties



- 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).

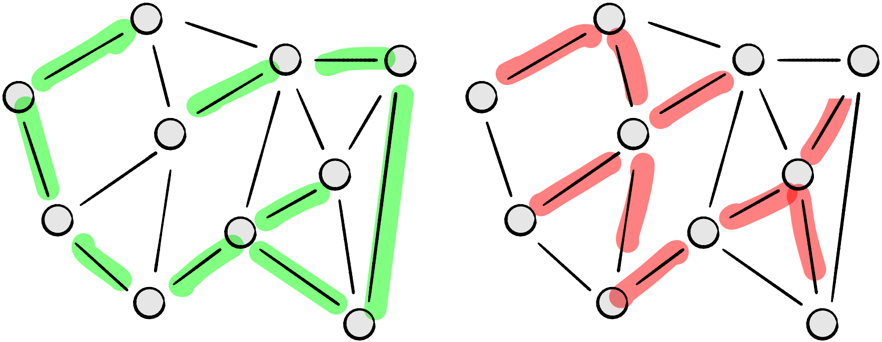
Spanning Trees

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.



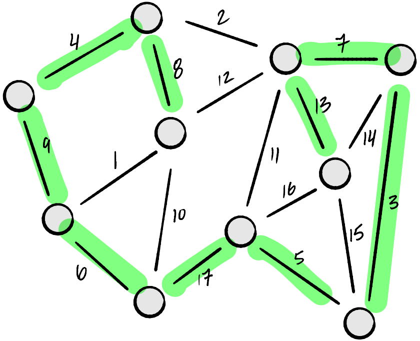
Many Spanning Trees

The same graph can have many spanning trees.



Spanning Tree Cost

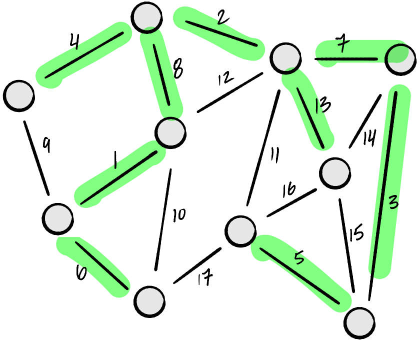
If $G = (V, E, \omega)$ is a weighted undirected graph, the **cost** (or **weight**) of a spanning tree is the total weight of the edges in the spanning tree.



Cost: $8+4+9+6+17+5+3+7+13 = 72$

Spanning Tree Cost

Different spanning trees of the same graph can have different costs.



Cost: 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 13 = 49

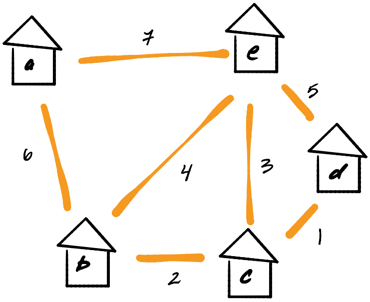
Minimum Spanning Tree

- 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.

Exercise

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

Today's Problem



- Choose a set of dirt roads to pave so that:
  - can get between any two buildings only on paved roads;
  - total cost is minimized.

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

MSTs in Data Science?

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

DSC 40B

Theoretical Foundations II

Lecture 16 | Part 2

Prim's Algorithm

Building MSTs

- How do we build a MST efficiently?

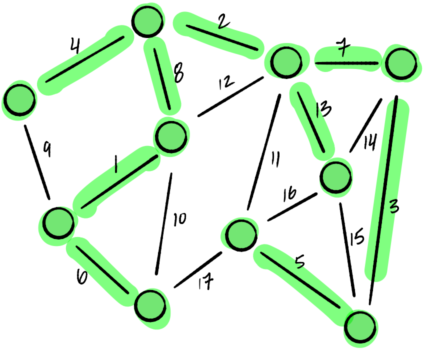
- We'll adopt a **greedy** approach.
  - Build a tree edge-by-edge.
  - At every step, doing what looks best at the moment.

- This strategy isn't guaranteed to work in all of life's situations, but it works for building MSTs.

Two Greedy Approaches

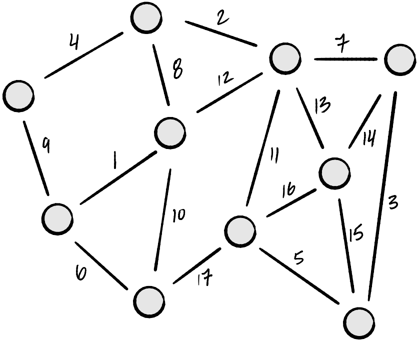
- We’ll look at two greedy algorithms:
  - Today: Prim’s Algorithm
  - Next time: Kruskal’s Algorithm
- Differ in the order in which edges are added to tree.
- Also differ in time complexity.

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 of 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 of smallest weight
- **Is this guaranteed to work?** Yes, as we'll see.

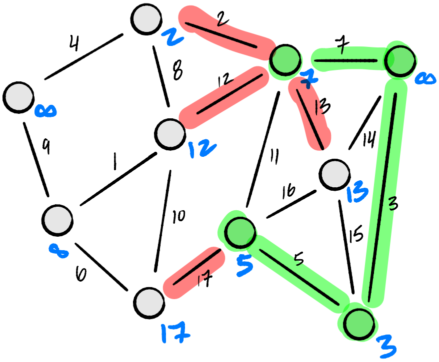
Prim’s Algorithm, Equivalently

- 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 aren't already in 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))
```

DSC 40B

Theoretical Foundations II

Lecture 16 | Part 3

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.

Time Complexity

$\Theta(V \log V + E \log V)$

```
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
```

$\Theta(V)$

V iterations

$O(V \log V)$

iters: $\Theta(E)$

$O(\log V)$ to run once

runs at most E times

tot: $O(E \log V)$

Time Complexity

- Using a **binary heap**...
- Overall: $\Theta(V \log V + E \log V)$.
- Since graph is assumed connected, $E = \Omega(V)$.
- 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[1].
  - `.decrease_priority()` takes $O(1)$ time.

[1] Amortized.

Time Complexity

```
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
```

$O(V \log V)$

$\Theta(E)$

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: $\Theta(E \log E) = \Theta(V^2 \log V)$.
- But Fibonacci heaps are **hard** to implement; have large constants.
- Binary heaps used more in practice despite complexity.

DSC 40B

Theoretical Foundations II

Lecture 16 | Part 4

Correctness of Prim's Algorithm

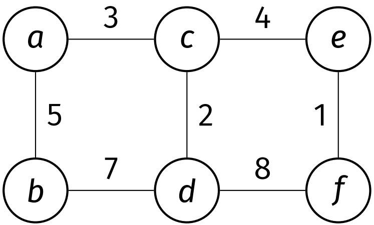
Being Greedy

- At every step, we add the lightest edge.
- Is this “safe”?

Being Greedy

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

Promising Subtrees



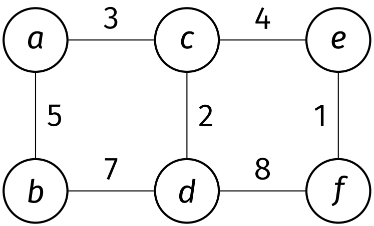
- Let $G = (V, E, \omega)$ be a weighted graph.
- A subgraph $T' = (V', E')$ is **promising** if it is "part" of some MST.
  - That is, it is an "MST in progress"
  - Not necessarily a tree!
- That is, there exists an MST $T = (V, E_{\text{mst}})$ such that $E' \subset E_{\text{mst}}$.
- Hint: a "promising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

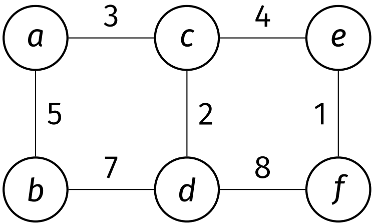
We'll show each new edge results in a larger promising subtree. Eventually the promising subtree becomes a full MST.

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 $T' + e$.