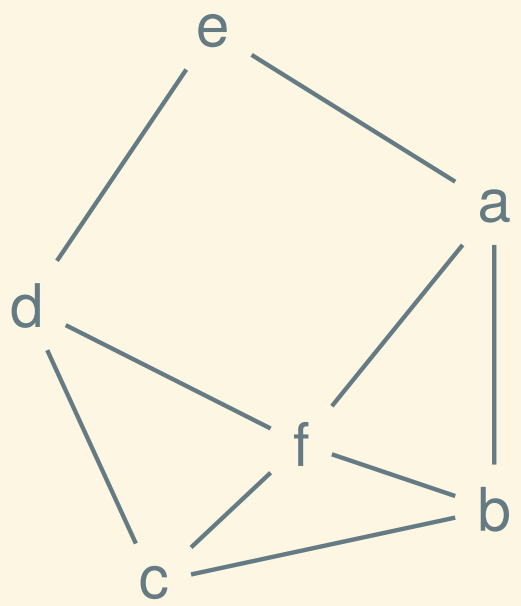
Minimum Spanning Tree

CS 214, Fall 2019

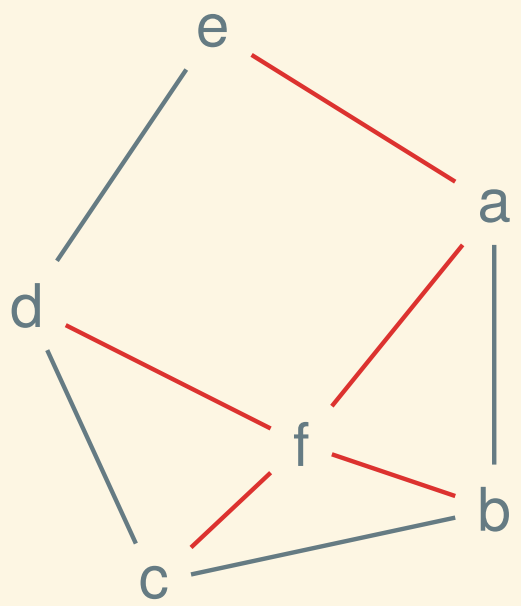
Definition: spanning tree

For a connected component of a graph, a *spanning tree* is a cycle-free subset of edges that connects every vertex:



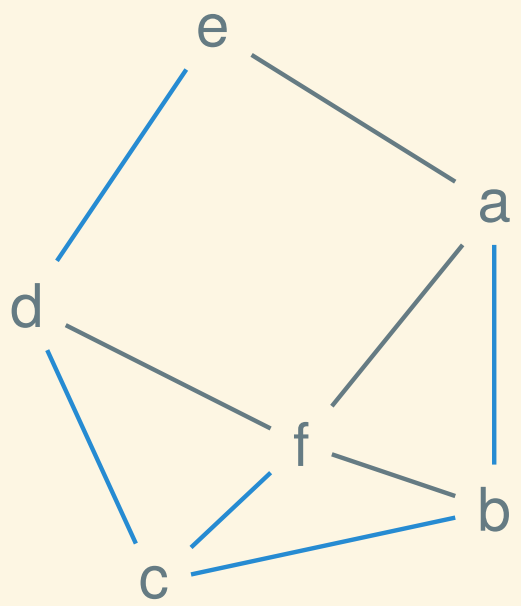
Definition: spanning tree

For a connected component of a graph, a *spanning tree* is a cycle-free subset of edges that connects every vertex:



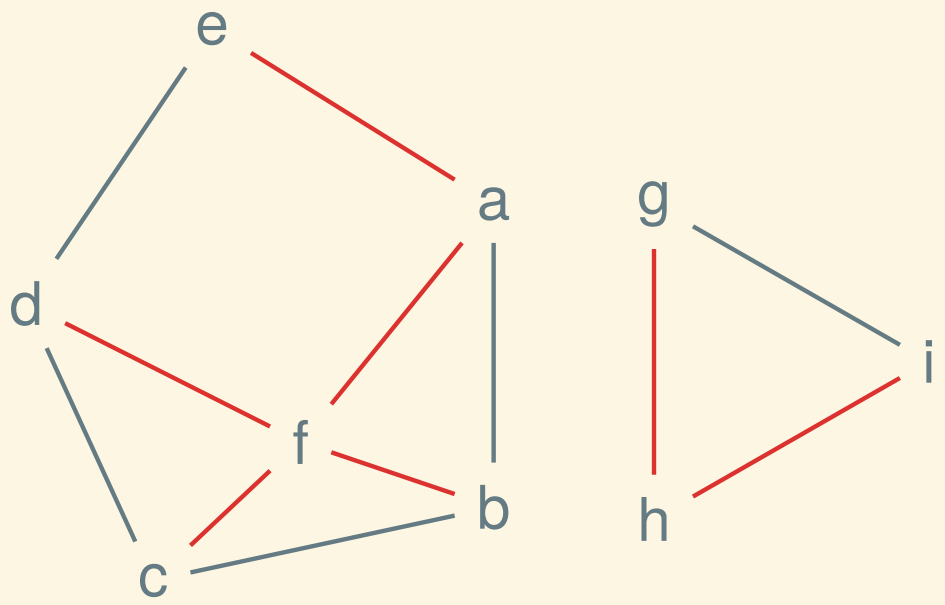
Definition: spanning tree

For a connected component of a graph, a *spanning tree* is a cycle-free subset of edges that connects every vertex:



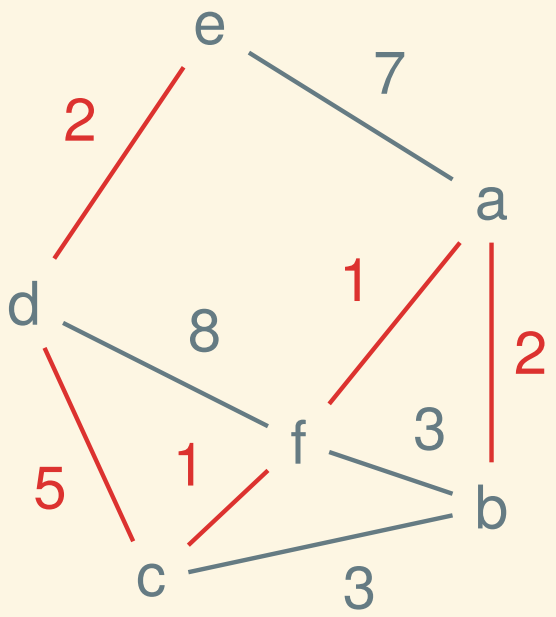
Definition: spanning forest

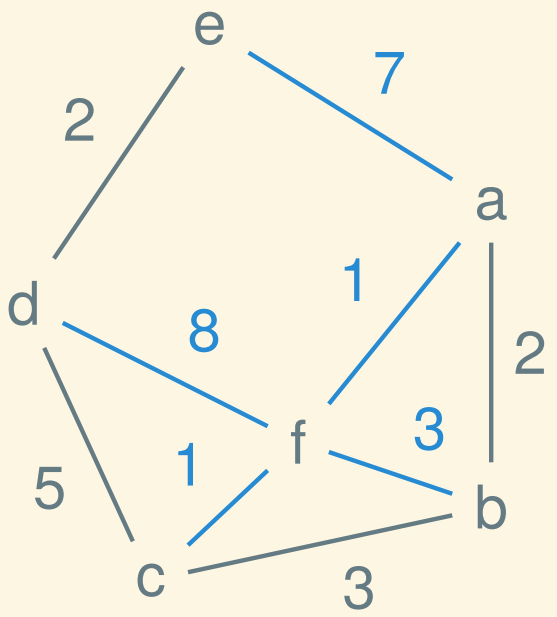
If a graph has multiple components then each will have its own spanning tree, forming a spanning forest:



Definition: minimal spanning tree

In a weighted graph, a spanning tree (or forest) is *minimal* if the sum of its weights is minimal over all possible spanning trees:





Computing an MST

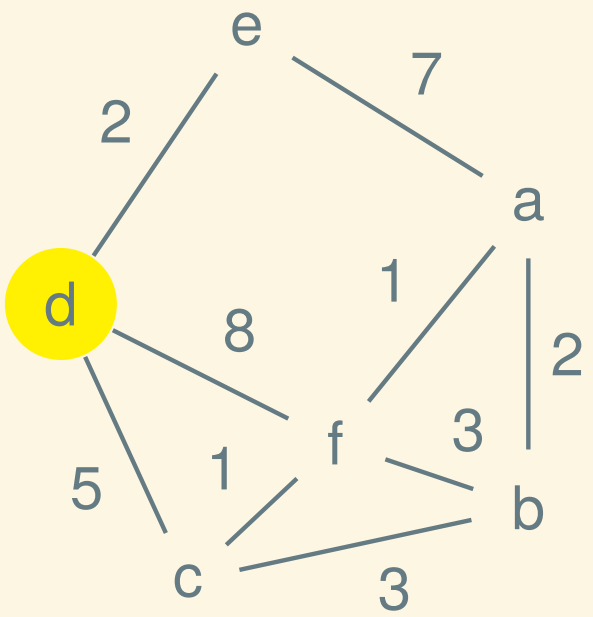
It's surprisingly easy—there are two simple, greedy algorithms:

- Prim's
- Kruskal's

Prim's algorithm

Build a tree edge-by-edge, as follows:

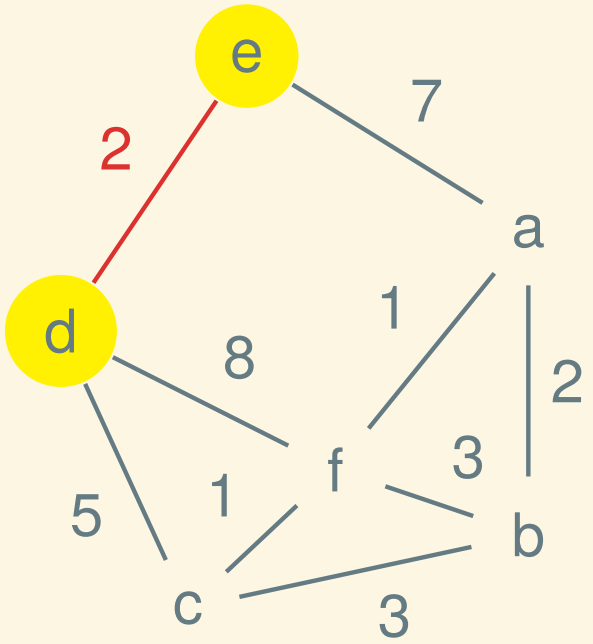
1. Start the tree at any vertex
2. Find the smallest edge connecting a tree vertex to a non-tree vertex, and add it to the tree
3. Repeat until all vertices are in the tree



Prim's algorithm

Build a tree edge-by-edge, as follows:

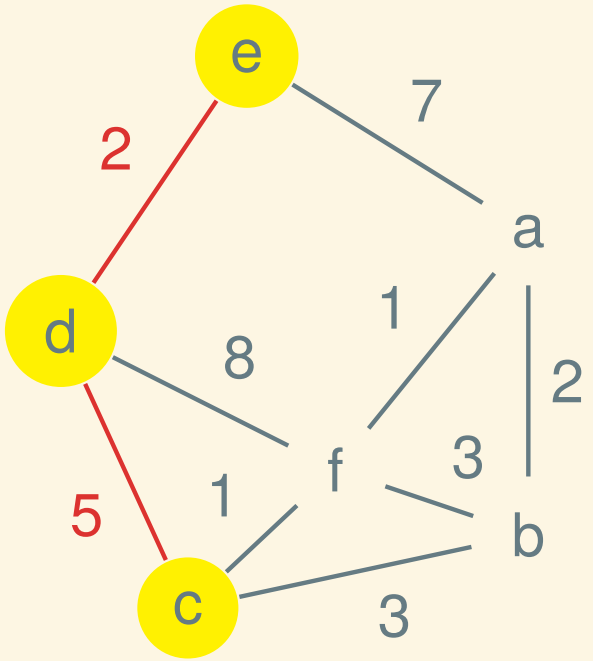
1. Start the tree at any vertex
2. Find the smallest edge connecting a tree vertex to a non-tree vertex, and add it to the tree
3. Repeat until all vertices are in the tree



Prim's algorithm

Build a tree edge-by-edge, as follows:

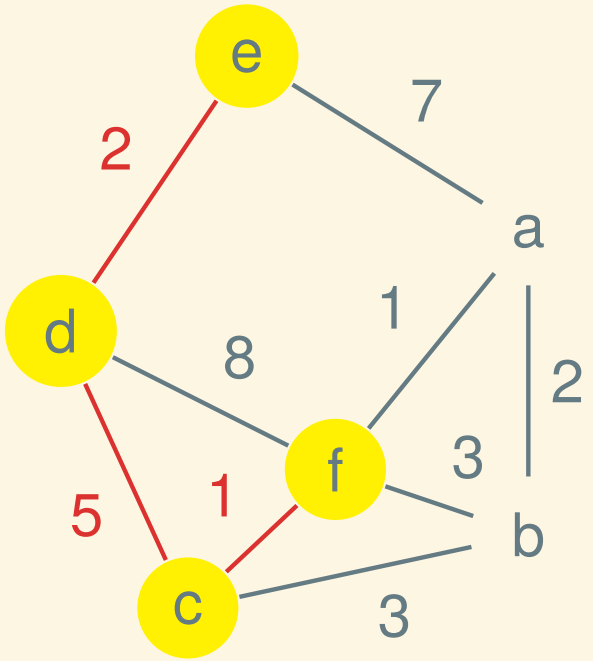
1. Start the tree at any vertex
2. Find the smallest edge connecting a tree vertex to a non-tree vertex, and add it to the tree
3. Repeat until all vertices are in the tree



Prim's algorithm

Build a tree edge-by-edge, as follows:

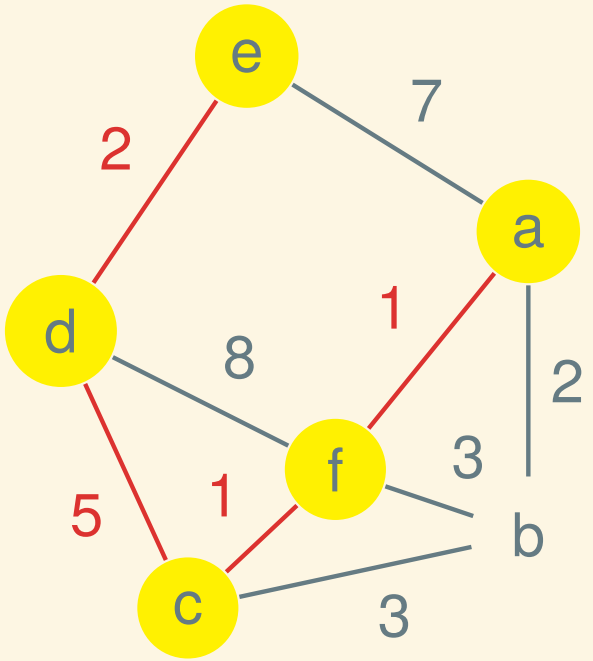
1. Start the tree at any vertex
2. Find the smallest edge connecting a tree vertex to a non-tree vertex, and add it to the tree
3. Repeat until all vertices are in the tree



Prim's algorithm

Build a tree edge-by-edge, as follows:

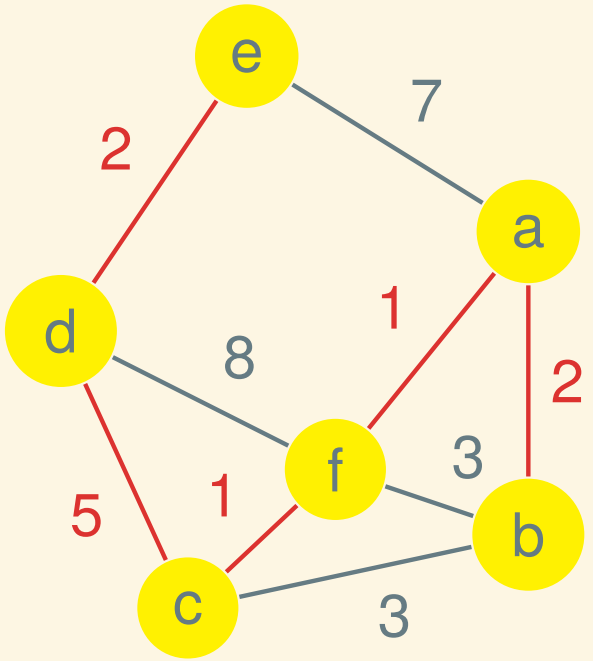
1. Start the tree at any vertex
2. Find the smallest edge connecting a tree vertex to a non-tree vertex, and add it to the tree
3. Repeat until all vertices are in the tree



Prim's algorithm

Build a tree edge-by-edge, as follows:

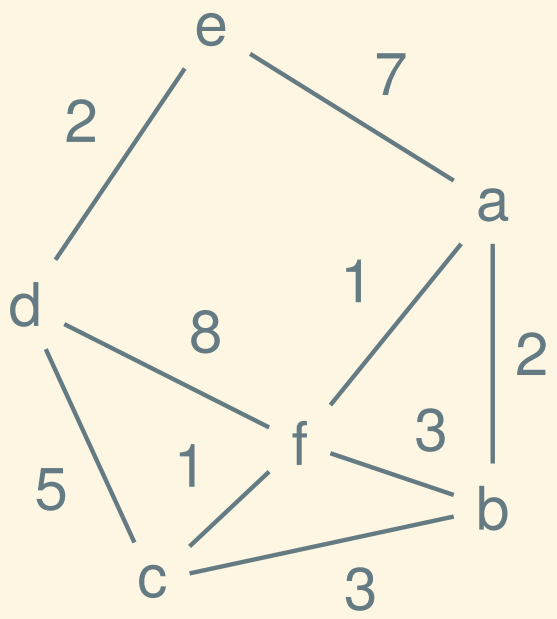
1. Start the tree at any vertex
2. Find the smallest edge connecting a tree vertex to a non-tree vertex, and add it to the tree
3. Repeat until all vertices are in the tree



Kruskal's algorithm

Build several trees and join them, as follows:

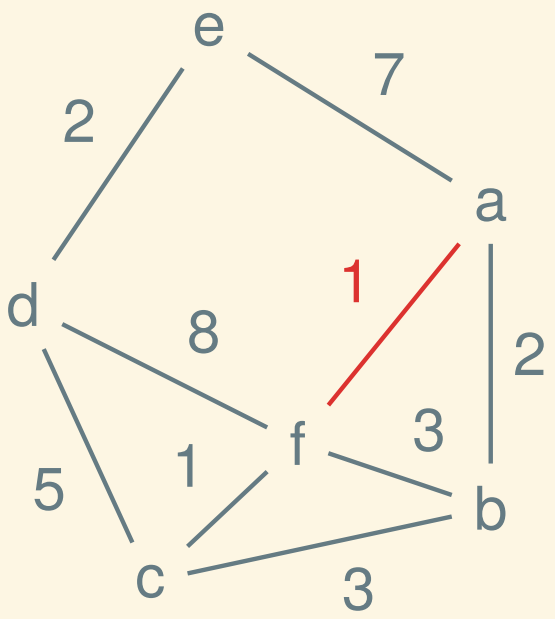
1. Start with a trivial tree at every vertex
2. Consider the edges in order from smallest to largest
3. When an edge would join two separate trees, use it combine them into one tree



Kruskal's algorithm

Build several trees and join them, as follows:

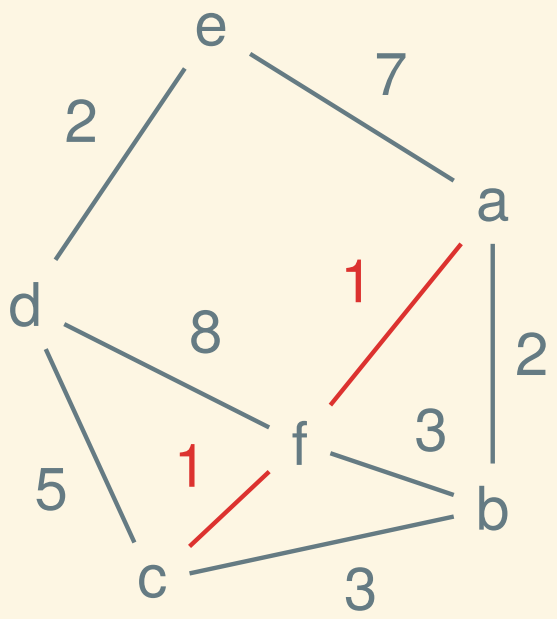
1. Start with a trivial tree at every vertex
2. Consider the edges in order from smallest to largest
3. When an edge would join two separate trees, use it combine them into one tree



Kruskal's algorithm

Build several trees and join them, as follows:

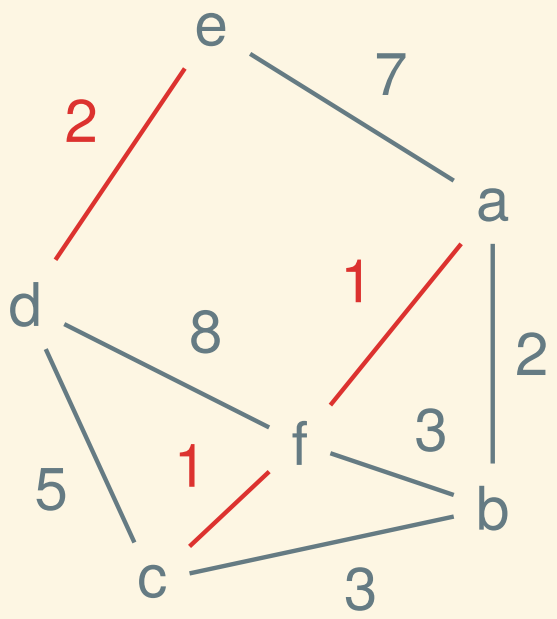
1. Start with a trivial tree at every vertex
2. Consider the edges in order from smallest to largest
3. When an edge would join two separate trees, use it combine them into one tree



Kruskal's algorithm

Build several trees and join them, as follows:

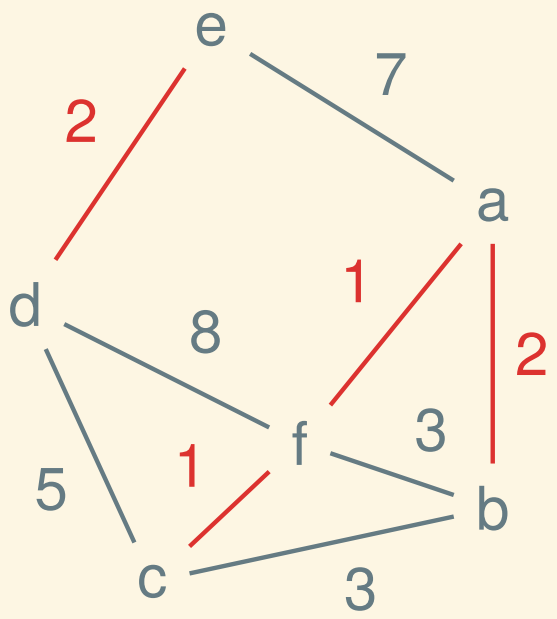
1. Start with a trivial tree at every vertex
2. Consider the edges in order from smallest to largest
3. When an edge would join two separate trees, use it combine them into one tree



Kruskal's algorithm

Build several trees and join them, as follows:

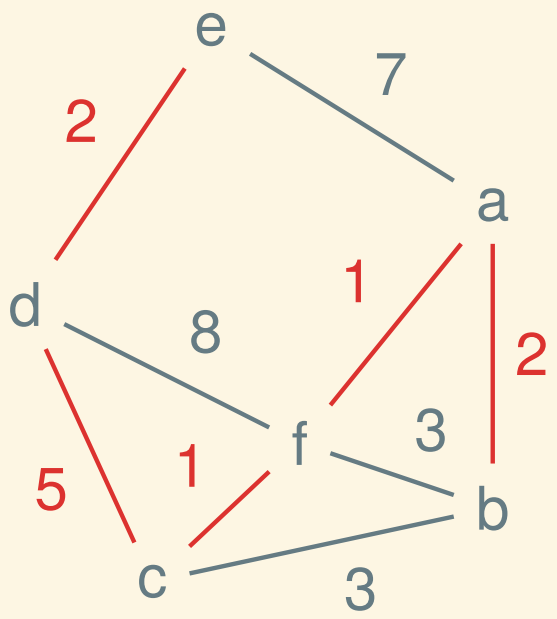
1. Start with a trivial tree at every vertex
2. Consider the edges in order from smallest to largest
3. When an edge would join two separate trees, use it combine them into one tree



Kruskal's algorithm

Build several trees and join them, as follows:

1. Start with a trivial tree at every vertex
2. Consider the edges in order from smallest to largest
3. When an edge would join two separate trees, use it combine them into one tree



Implementing Kruskal's algorithm

We need a way to keep track of the disjoint trees

Disjoint Sets ADT (aka Union-Find), take 1

Looks like: 0 {1 2 5} {3 7} 4 6

```
interface UNION_FIND:
    def len(self) -> nat?
    def union(self, p: nat?, q: nat?) -> VoidC
    def same_set?(self, p: nat?, q: nat?) -> bool?
```

Behavior:

- `d.len()` returns the total number of objects
- `d.union(p, q)` causes p and q's sets to be joined together
- `d.same_set?(p, q)` reports whether p and q are now in the same set

Kruskal's algorithm using disjoint sets

Input: A weighted graph *graph* of n vertices
Output: A minimum spanning forest *forest* (represented as a graph)

uf $\leftarrow$ a new union-find universe with n objects;
forest $\leftarrow$ a graph of n vertices and 0 edges;

for each edge (u, v) in increasing order of weight w do
    if ¬`sameSet(`*uf, u, v*`)` then
        `union(`*uf, u, v*`)`;
        `addEdge(`*forest, u, v*`)`
    end
end

Implementing union-find

Goal

Efficient data structure for union-find:

- `union` commands and `same_set?` queries can be interleaved
- number of operations m can be huge
- number of objects n can be huge

We're also going think about efficiency in terms of Kruskal's algorithm: $\mathcal{O}(E \log E + ET_{sameSet} + VT_{union})$

A twist

It turns out that the `same_set?` operation isn't exactly the right one, so in practice the ADT offers a different operation, `find`.

Disjoint Sets ADT (aka Union-Find), for real

Each set has one, distinguished representative object:

0 {1 2 5} {3 7} 4 6

```
interface UNION_FIND:
    def len(self) -> nat?
    def union(self, p: nat?, q: nat?) -> VoidC
    def find(self, p: nat?) -> nat?
```

Behavior:

- `d.union(p, q)` causes p and q's sets to be joined together
- `d.find(p)` reports p's representative

Kruskal's algorithm using disjoint sets

Input: A weighted graph *graph* of n vertices
Output: A minimum spanning forest *forest* (represented as a graph)

uf $\leftarrow$ a new union-find universe with n objects;
forest $\leftarrow$ a graph of n vertices and 0 edges;

for each edge (u, v) in increasing order of weight w do
    if `find(`*uf*`,` *u*`)` $\neq$ `find(`*uf*`,` *v*`)` then
        `union(`*uf*`,` *u*`,` *v*`);`
        `addEdge(`*forest*`,` *u*`,` *v*`)`
    end
end

Union-Find example

d = 0 1 2 3 4 5 6 7

Union-Find example

d = 0 1 2 3 4 5 6 7

`d.find(0)` $\Rightarrow$

Union-Find example

d = 0 1 2 3 4 5 6 7

`d.find(0)` $\Rightarrow$ `0`

Union-Find example

d = 0 1 2 3 4 5 6 7

`d.find(0)` $\Rightarrow$ `0` `d.find(1)` $\Rightarrow$

Union-Find example

d = 0 1 2 3 4 5 6 7

`d.find(0)` $\Rightarrow$ `0` `d.find(1)` $\Rightarrow$ `1`

Union-Find example

d = 0 1 2 3 4 5 6 7

`d.find(0)` $\Rightarrow$ `0` `d.find(1)` $\Rightarrow$ `1`

```
d.union(2, 5)
```

Union-Find example

d = 0 1 2 3 4 5 6 7

`d.find(0)` ⇒ `0` `d.find(1)` ⇒ `1`

```
d.union(2, 5)
```

d = 0 1 {2 5} 3 4 6 7

```
d.find(2)
```
⇒

Union-Find example

d = 0 1 2 3 4 5 6 7

`d.find(0)` $\Rightarrow$ `0` `d.find(1)` $\Rightarrow$ `1`

```
d.union(2, 5)
```

d = 0 1 {2 5} 3 4 6 7

`d.find(2)` $\Rightarrow$ `2`

Union-Find example

d = 0 1 2 3 4 5 6 7

`d.find(0)` ⇒ 0 `d.find(1)` ⇒ 1

```
d.union(2, 5)
```

d = 0 1 {2 5} 3 4 6 7

`d.find(2)` ⇒ 2 `d.find(5)` ⇒

Union-Find example

d = 0 1 2 3 4 5 6 7

`d.find(0)` ⇒ `0` `d.find(1)` ⇒ `1`

```
d.union(2, 5)
```

d = 0 1 {2 5} 3 4 6 7

`d.find(2)` ⇒ `2` `d.find(5)` ⇒ `2`

Union-Find example

d = 0 1 2 3 4 5 6 7

`d.find(0)` ⇒ 0 `d.find(1)` ⇒ 1

```
d.union(2, 5)
```

d = 0 1 {2 5} 3 4 6 7

`d.find(2)` ⇒ 2 `d.find(5)` ⇒ 2

```
d.union(1, 2)
```

Union-Find example

d = 0 1 2 3 4 5 6 7

`d.find(0)` ⇒ 0 `d.find(1)` ⇒ 1

`d.union(2, 5)`

d = 0 1 {2 5} 3 4 6 7

`d.find(2)` ⇒ 2 `d.find(5)` ⇒ 2

`d.union(1, 2)`

d = 0 {1 2 5} 3 4 6 7

Union-Find example

d = 0 1 2 3 4 5 6 7

`d.find(0)` ⇒ 0 `d.find(1)` ⇒ 1

`d.union(2, 5)`

d = 0 1 {2 5} 3 4 6 7

`d.find(2)` ⇒ 2 `d.find(5)` ⇒ 2

`d.union(1, 2)`

d = 0 {1 2 5} 3 4 6 7

`d.union(3, 7)`

Union-Find example

d = 0 1 2 3 4 5 6 7

`d.find(0)` ⇒ 0 `d.find(1)` ⇒ 1

`d.union(2, 5)`

d = 0 1 {2 5} 3 4 6 7

`d.find(2)` ⇒ 2 `d.find(5)` ⇒ 2

`d.union(1, 2)`

d = 0 {1 2 5} 3 4 6 7

`d.union(3, 7)`

d = 0 {1 2 5} {3 7} 4 6

Union-Find example

d = 0 1 2 3 4 5 6 7

`d.find(0)` ⇒ 0 `d.find(1)` ⇒ 1

`d.union(2, 5)`

d = 0 1 {2 5} 3 4 6 7

`d.find(2)` ⇒ 2 `d.find(5)` ⇒ 2

`d.union(1, 2)`

d = 0 {1 2 5} 3 4 6 7

`d.union(3, 7)`

d = 0 {1 2 5} {3 7} 4 6

Next: making Union-Find efficient