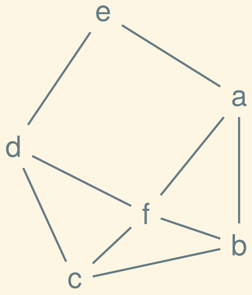
Minimum Spanning Tree

EECS 214, Fall 2017

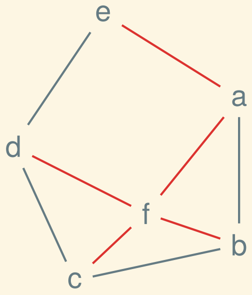
Definition: spanning tree

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



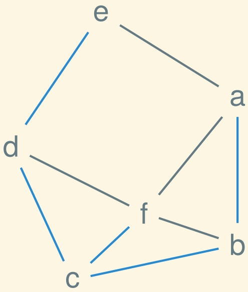
Definition: spanning tree

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



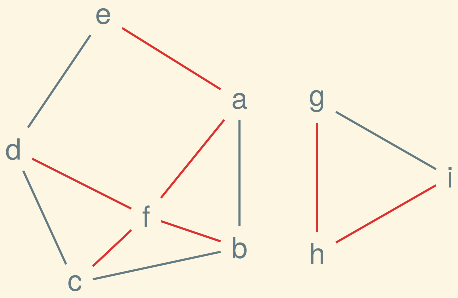
Definition: spanning tree

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



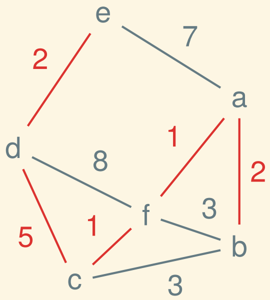
Definition: spanning forest

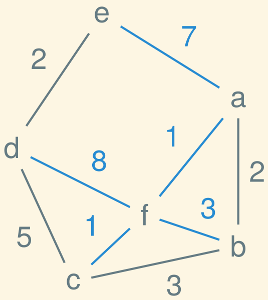
If a graph has multiple components then each will have its own spanning tree, forming a spanning forest:



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

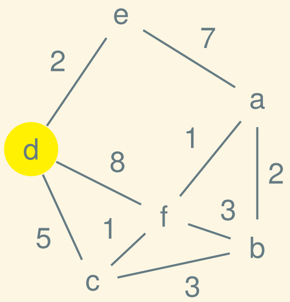
It's surprisingly easy—there are two simple, greedy algorithms:

- Prim's
- Kruskal's

Prim's algorithm

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

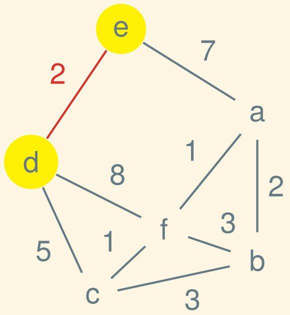
1. Start the tree at any vertex
2. Find the smallest edge connecting a tree vertex to a non-tree vertex, and add it to the tree
3. Repeat until all vertices are in the tree



Prim's algorithm

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

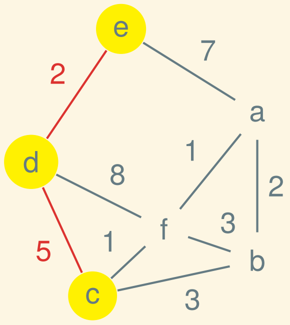
1. Start the tree at any vertex
2. Find the smallest edge connecting a tree vertex to a non-tree vertex, and add it to the tree
3. Repeat until all vertices are in the tree



Prim's algorithm

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

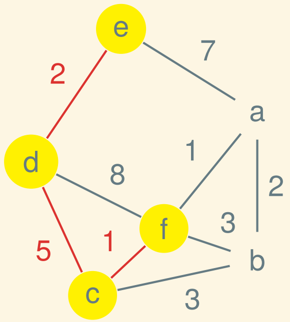
1. Start the tree at any vertex
2. Find the smallest edge connecting a tree vertex to a non-tree vertex, and add it to the tree
3. Repeat until all vertices are in the tree



Prim's algorithm

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

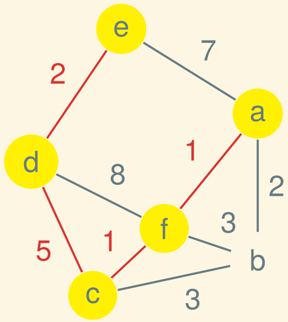
1. Start the tree at any vertex
2. Find the smallest edge connecting a tree vertex to a non-tree vertex, and add it to the tree
3. Repeat until all vertices are in the tree



Prim's algorithm

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

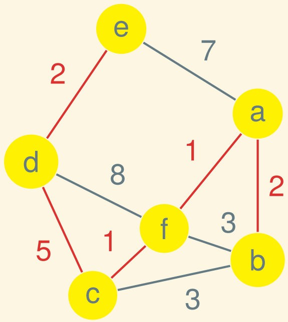
1. Start the tree at any vertex
2. Find the smallest edge connecting a tree vertex to a non-tree vertex, and add it to the tree
3. Repeat until all vertices are in the tree



Prim's algorithm

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

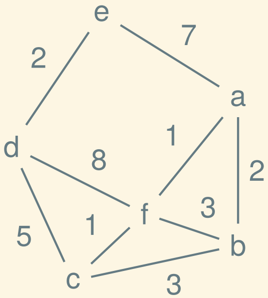
1. Start the tree at any vertex
2. Find the smallest edge connecting a tree vertex to a non-tree vertex, and add it to the tree
3. Repeat until all vertices are in the tree



Kruskal’s algorithm

Build several trees and join them, as follows:

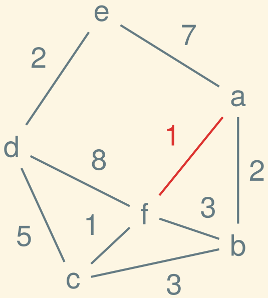
1. Start with a trivial tree at every vertex
2. Consider the edges in order from smallest to largest
3. When an edge would join two separate trees, use it combine them into one tree



Kruskal's algorithm

Build several trees and join them, as follows:

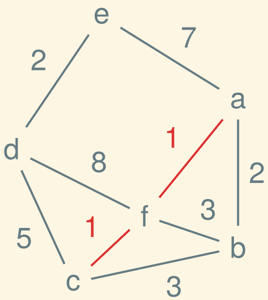
1. Start with a trivial tree at every vertex
2. Consider the edges in order from smallest to largest
3. When an edge would join two separate trees, use it combine them into one tree



Kruskal's algorithm

Build several trees and join them, as follows:

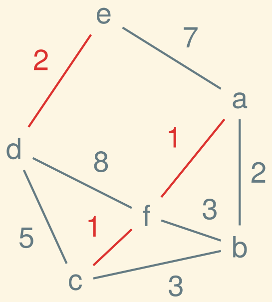
1. Start with a trivial tree at every vertex
2. Consider the edges in order from smallest to largest
3. When an edge would join two separate trees, use it combine them into one tree



Kruskal’s algorithm

Build several trees and join them, as follows:

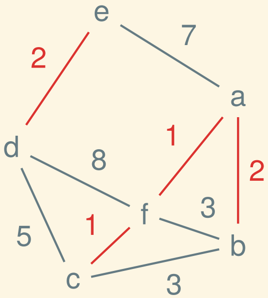
1. Start with a trivial tree at every vertex
2. Consider the edges in order from smallest to largest
3. When an edge would join two separate trees, use it combine them into one tree



Kruskal's algorithm

Build several trees and join them, as follows:

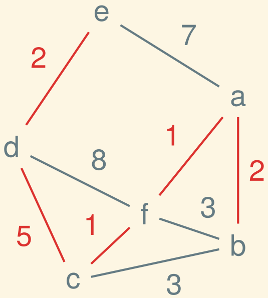
1. Start with a trivial tree at every vertex
2. Consider the edges in order from smallest to largest
3. When an edge would join two separate trees, use it combine them into one tree



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)

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

Signatures:

- *union*(DisjointSets, Nat, Nat): Void
- *find*(DisjointSets, Nat): Nat

Behavior:

- *union*(d, p, q) causes p and q's sets to be joined together
- *find*(d, p) returns a representative element that will be the same for every element of p's 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 find(uf, u) ≠ find(uf, v) then
        union(uf, u, v);
        addEdge(forest, u, v)
    end
end
```

Implementing union-find

Goal

Efficient data structure for union-find:

- find and union commands can be interleaved
- number of operations m can be huge
- number of objects n can be huge

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