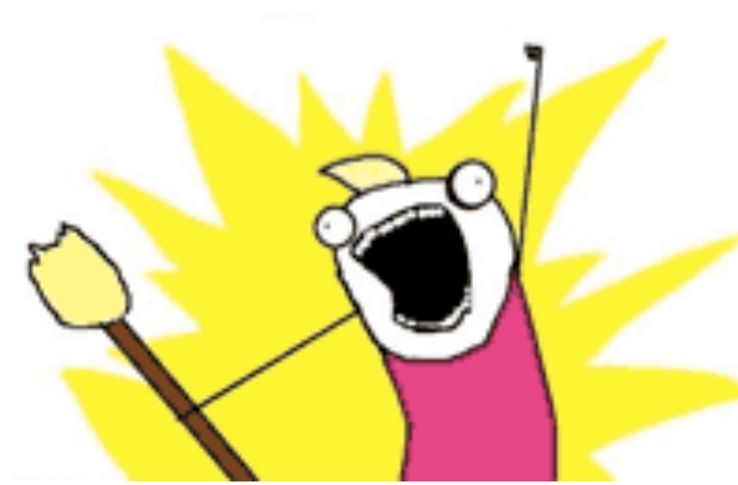
CSC263 Week 10

Announcements

Problem Set 5 is out (today)!

Due Tuesday (Dec 1)

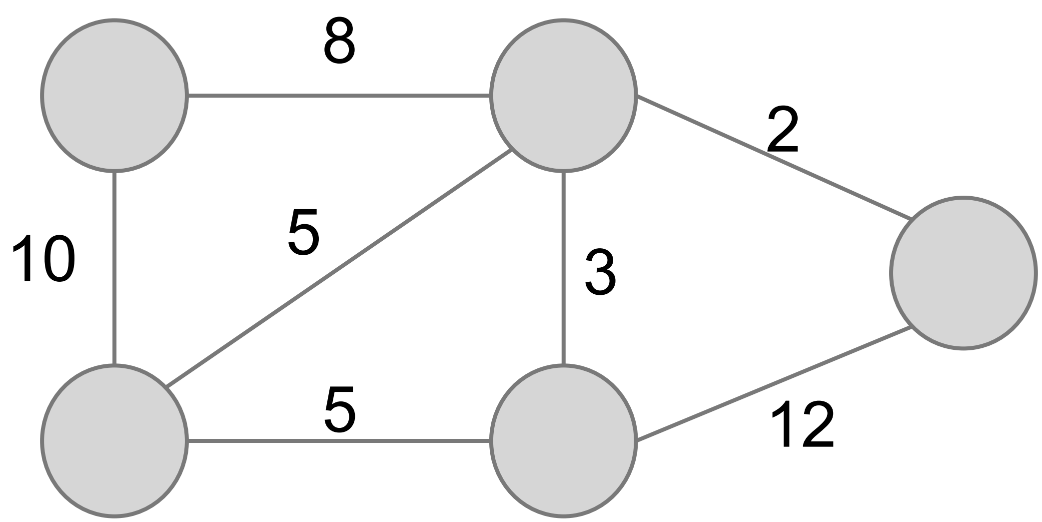


Minimum Spanning Trees

The Graph of interest today

A **connected undirected** **weighted** graph

G = (V, E) with weights **w(e)** for each e ∈ E



It has the **smallest** total weight

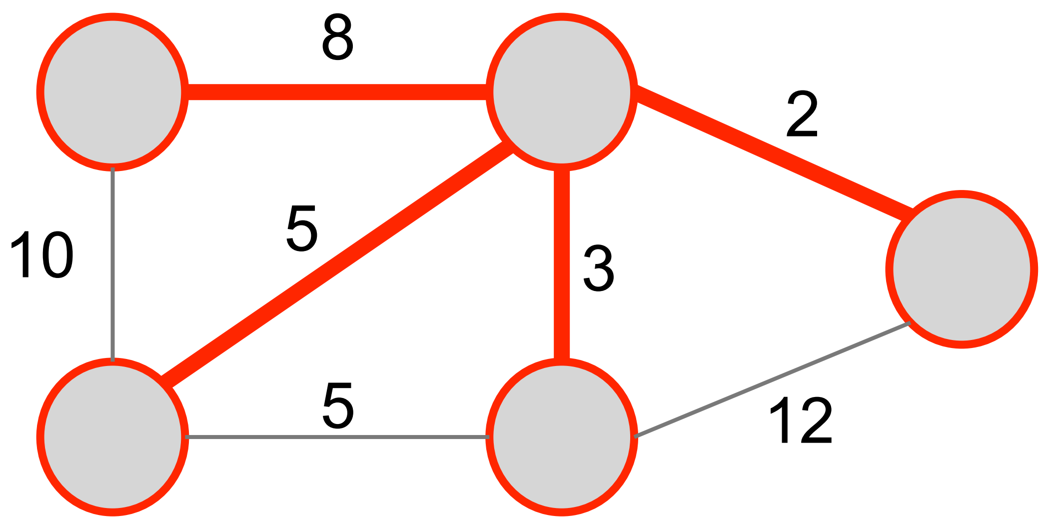
It covers **all** vertices in G

Minimum Spanning Tree

of graph G

It's a **connected**, **acyclic** subgraph

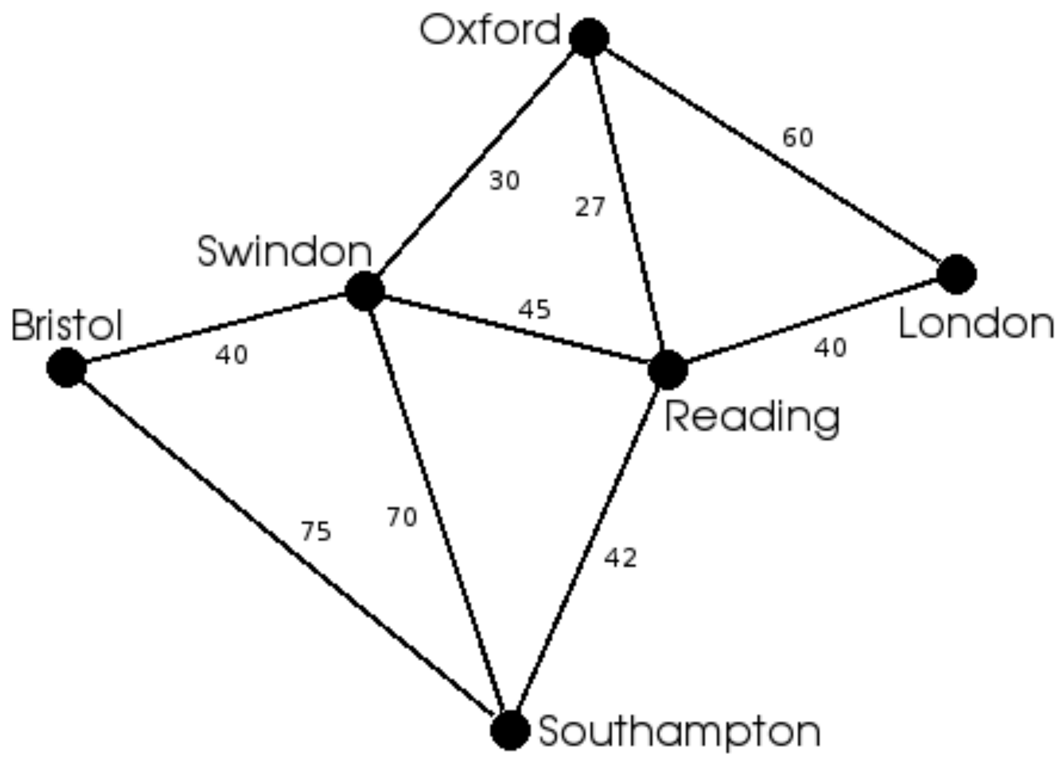
A Minimum Spanning Tree



May NOT be unique

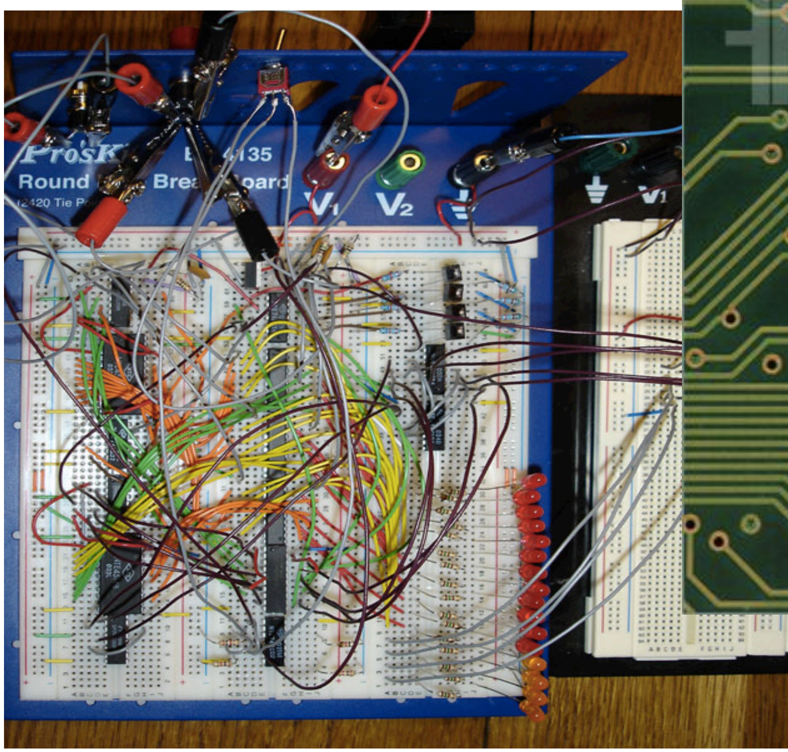
Applications of MST

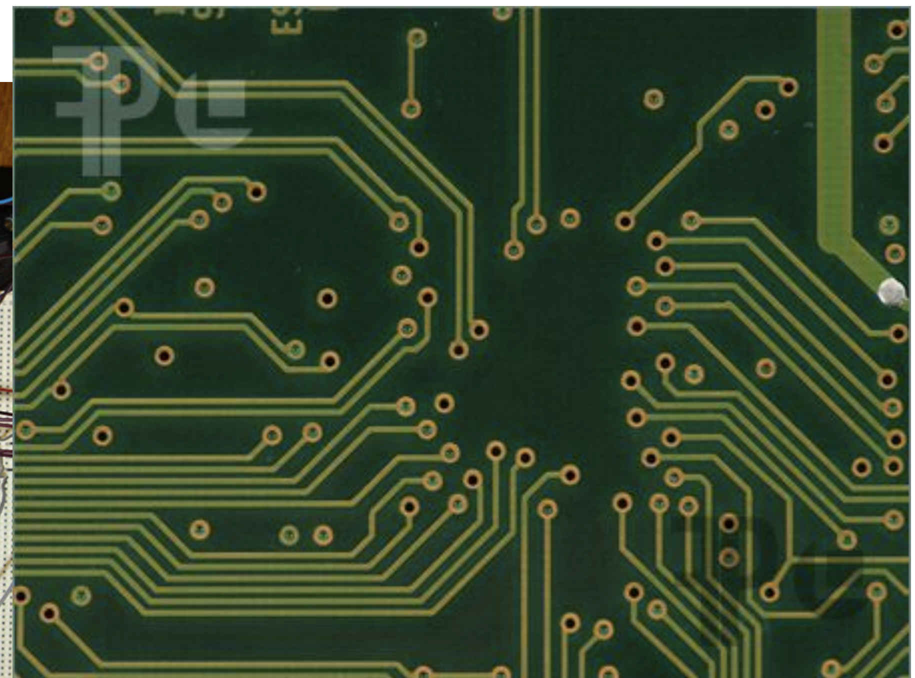
Build a road network that connects all towns and with the minimum cost.



Applications of MST

Connect all components with the least amount of wiring.





Other applications

- ➔Cluster analysis
- ➔Approximation algorithms for the “travelling salesman problem”
- ➔...

In order to understand

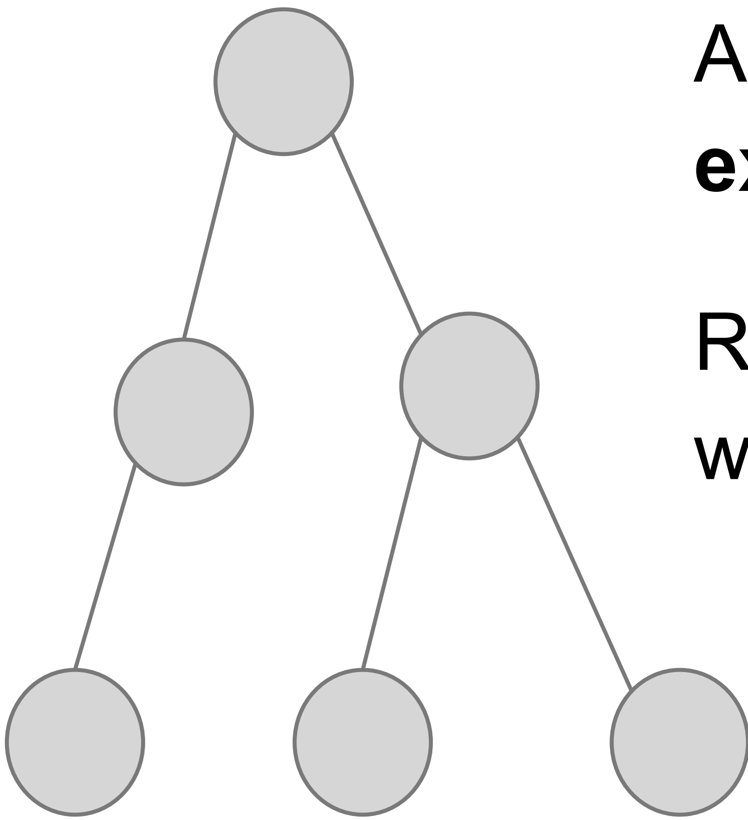
minimum spanning tree

we need to first understand

tree

Tree:

undirected connected acyclic graph



A tree **T** with **n** vertices has **exactly** ___n-1___ edges.

Removing one edge from T will ___disconnect the tree___

Adding one edge to T will ___create a cycle___.

The MST of a connected graph G = (V, E) has ____|V|____ vertices.

because "spanning"

The MST of a connected graph G = (V, E) has ____|V| - 1____ edges.

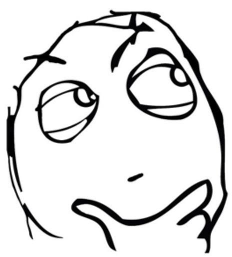
because "tree"

Now we are ready to talk about

algorithms

Idea #1

Start with **T = G.E**, then keep deleting edges until an MST remains.



Which sounds more efficient in terms of worst-case runtime?

Idea #2

Start with **empty** T, then keep adding edges until an MST is built.

Hint

A undirected simple graph G with **n** vertices can have at most

______________ edges.

$$\binom{n}{2} = \frac{n(n-1)}{2} \in \mathcal{O}(n^2)$$

Note: Here T is an edge set

Idea #1

Start with T = G.E, then keep deleting edges until an MST remains.

In worst-case, need to delete $O(|V|^2)$ edges *(n choose 2) - (n-1)*

Idea #2

In worst-case, need to add $O(|V|)$ edges

Start with **empty** T, then keep adding edges until an MST is built.

This is more efficient!

So, let's explore more of **Idea #2**,
i.e.,
building an MST by **adding** edges
one by one

i.e.,
we "grow" a tree

The generic growing algorithm

```
GENERIC-MST(G=(V, E, w)):
    T ← ∅
    while T is not a spanning tree:
        find a “safe” edge e
        T ← T ∪ {e}
    return T
```

|T| < |V|-1

What is a “safe” edge?

"Safe" means it keeps the **hope** of T growing into an MST.

"Safe" edge e for T

Assuming ***before*** *adding e,* ***T*** $\subseteq$ ***some MST****,*
edge **e** *is safe if* ***after*** *adding* **e***, still* ***T*** $\subseteq$ ***some MST***

If we make sure T is always a subset of some MST while we grow it, then eventually T will become an MST!

```
GENERIC-MST(G=(V, E, w)):
    T ← ∅
    while T is not a spanning tree:
        find a "safe" edge e
        T ← T ∪ {e}
    return T
```

Intuition

If we make sure the pieces we put together is always a subset of the real picture while we grow it, then eventually it will become the real picture!

The generic growing algorithm

```
GENERIC-MST(G=(V, E, w)):
    T ← ∅
    while T is not a spanning tree:
        find a "safe" edge e
        T ← T ∪ {e}
    return T
```

|T| < |V|-1

How to find a “safe” edge?

Two major algorithms we'll learn

➔Kruskal's algorithm

➔Prim's algorithm

They are both based on one theorem...

How to find a safe edge: The cut property

Let G=(V,E) be a connected, undirected graph.

X a subset of edges of G such that T contains X, where T is a minimum spanning tree of G. (So X is a forest and can be extended to a MST)

Let S be a connected component of (V,X). (So no edge in X crosses the cut S, V-S)

Among all edges crossing between S and V-S, let e be an edge of minimum weight.

Then some MST T' contains X+e (In other words, e is a safe edge.)

Basic outline of all MST algs:

Start with $G=(V,E,w)$
Let X be a set of edges, initially X is empty
Repeat until $|X| = |V|-1$:

1. Pick a connected component S of (V,X)
2. Let e be a lightest edge in E that crosses between S and $V-S$
3. Add e to X

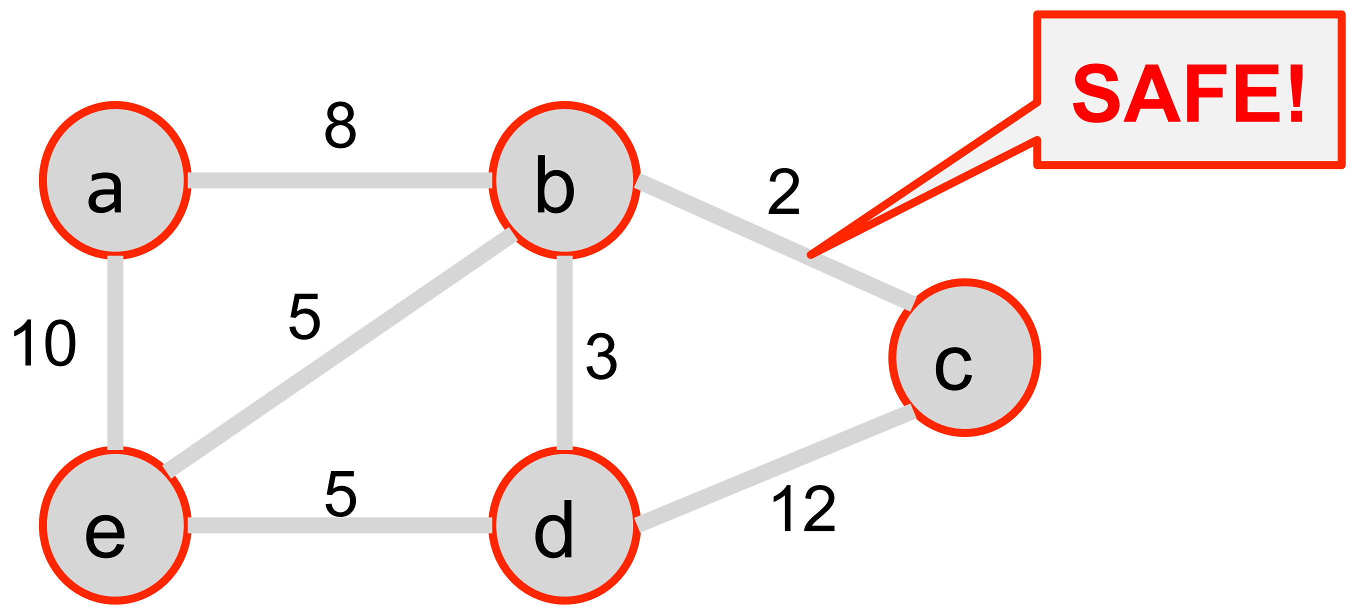
Basic outline of all MST algs:

Start with G=(V,E,w)
Let X be a set of edges, initially E is empty
Repeat until |X| = |V|-1:

1. Pick a connected component S of (V,X)
2. Let e be a lightest edge in E that crosses between S and V-S
3. Add e to X

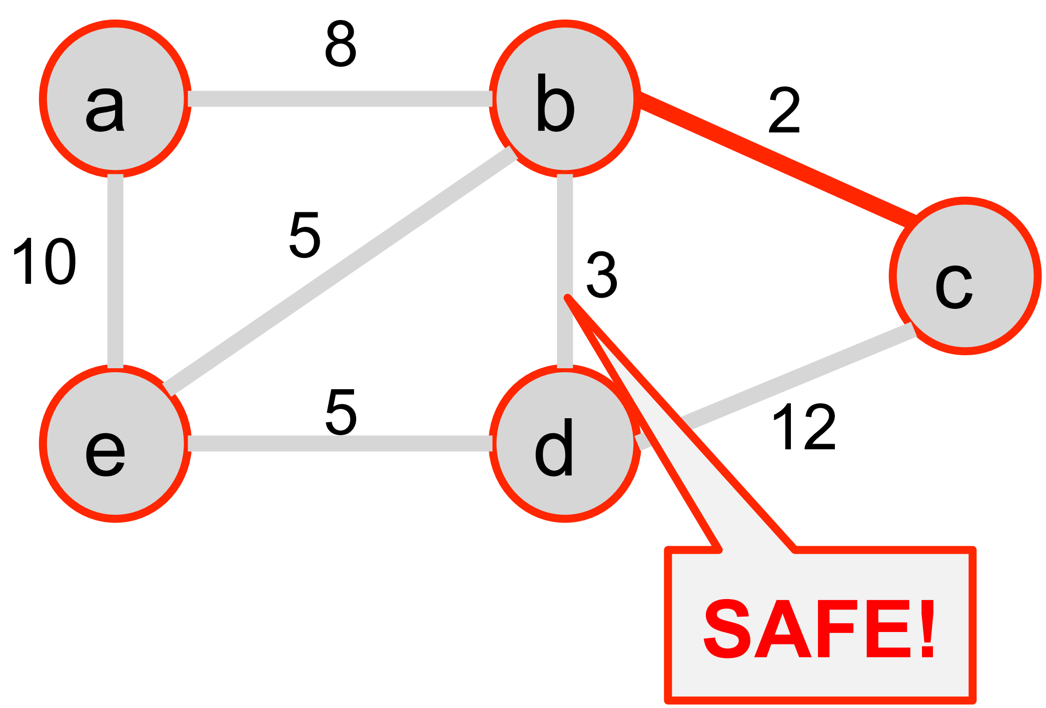
Prim: S starts off being a single vertex r, and in general S is the connected component containing r
Kruskal: choose S so that the length of e is minimum

Initially, **T** (red) is a subgraph with no edge,
each vertex is a connected component,
all edges are **crossing** components,
and the minimum weighted one is ...

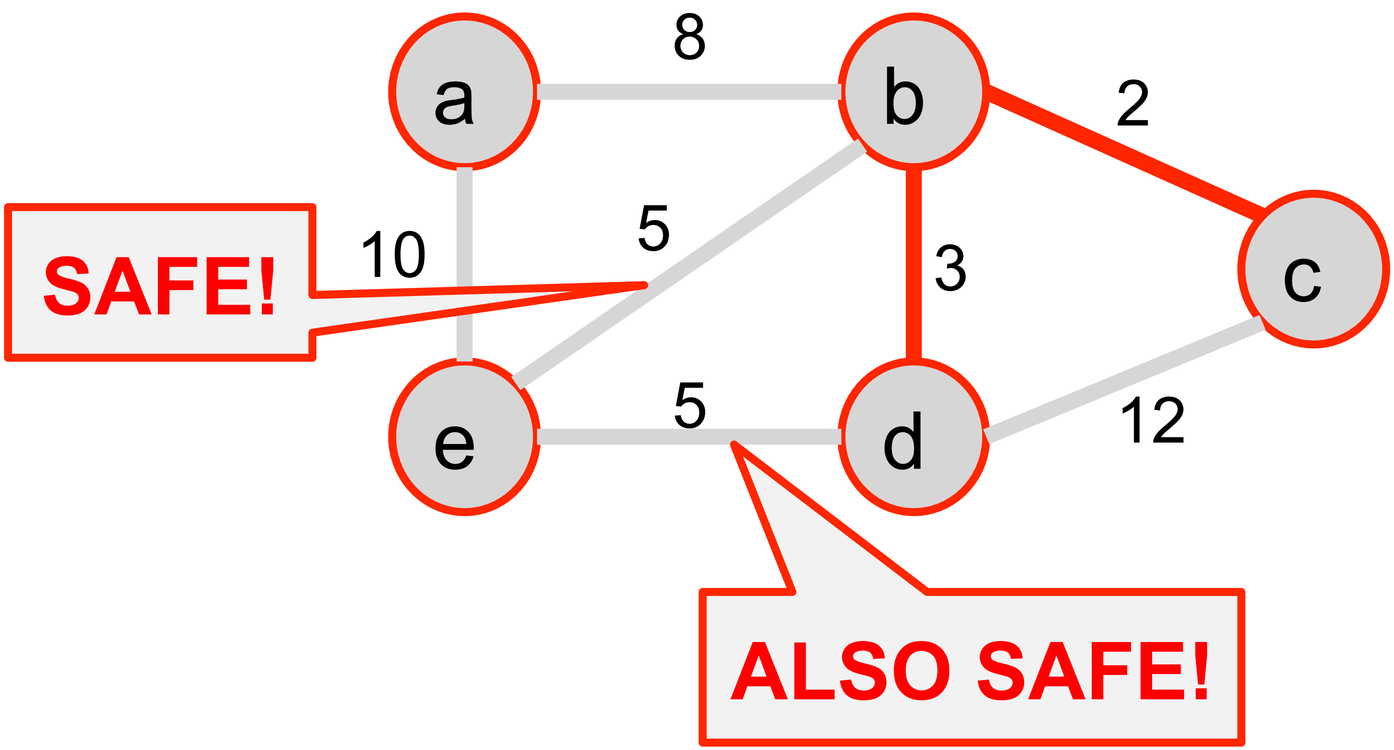


Now **b and c** in one connected component, each of the other vertices is a component, i.e., 4 components.

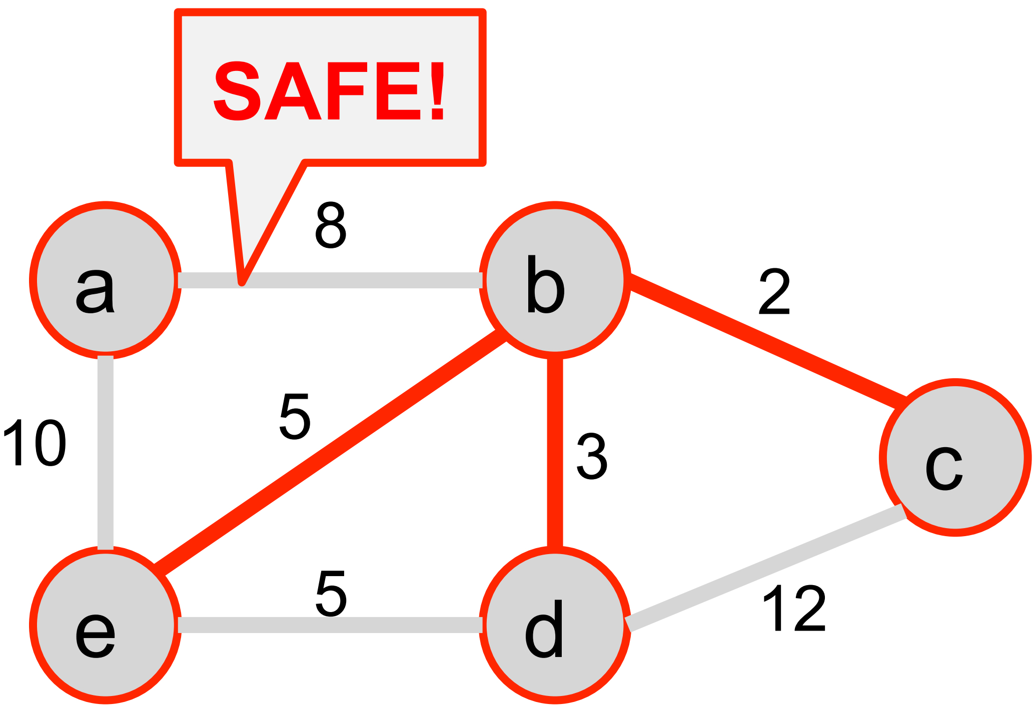
All gray edges are crossing components.



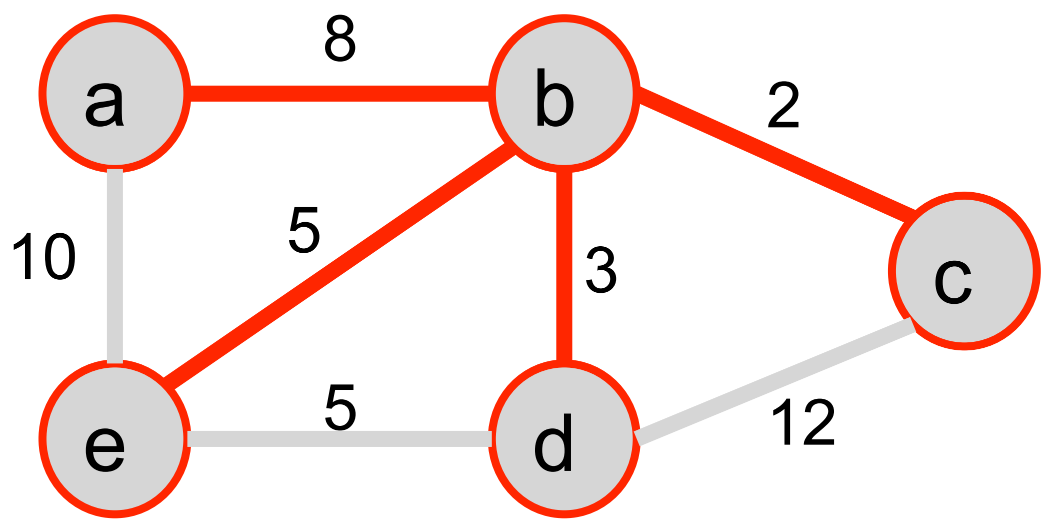
Now **b, c and d** are in one connected component, **a** and **e** each is a component.
(c, d) is **NOT** crossing components!



Now **b, c, d and e** are in one connected component, **a** is a component.
(a, e) and **(a, b)** are crossing components.



MST grown!



Two things that need to be worried about when actually implementing the algorithm

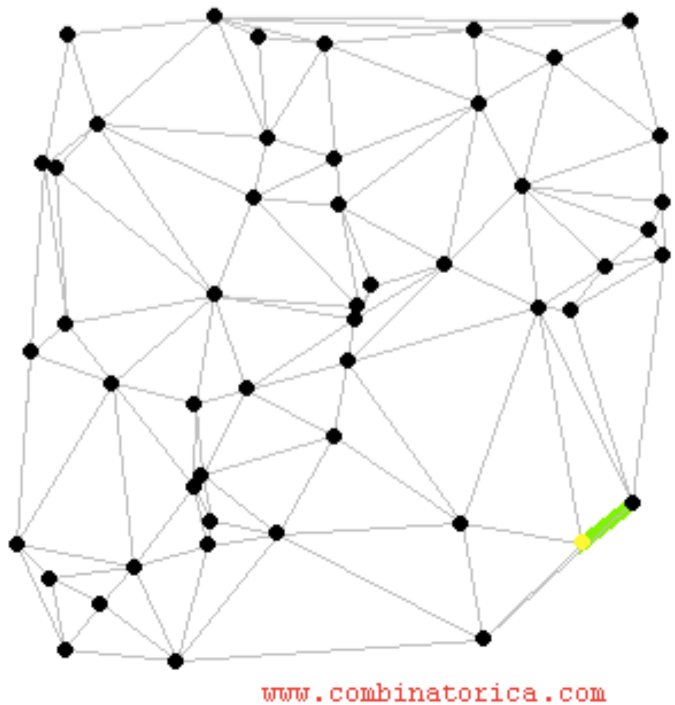
- ➔How to keep track of the **connected components**?
- ➔How to efficiently find the **minimum** weighted edge?

Kruskal's and **Prim's** basically use different **data structures** to do these two things.

Overview: Prim's and Kruskal's

	Keep track of connected components	**Find minimum weight edge**
Prim's	*Keep "one tree plus isolated vertices"*	*use priority queue ADT*
Kruskal's	*use "disjoint set" ADT*	*Sort all edges according to weight*

Prim's



Kruskal's

https://trendsofcode.files.wordpress.com/2014/09/dijkstra.gif

https://www.projectrhea.org/rhea/images/4/4b/Kruskal_Old_Kiwi.gif

Prim’s MST algorithm

Prim's algorithm: Idea

- ➔ Start from an arbitrary vertex as root
- ➔ Focus on growing **one** tree. This tree is one **component**; the cut is always (T,V-T) where T is the tree so far.)
- ➔ Choose a minimum weight edge among all edges that are **incident to the current tree** (**edges crossing the cut**)
- ➔ How to get that minimum? Store all candidate vertices in a **Min-Priority Queue** whose key is the weight of the **crossing** edge (incident to tree).

```
PRIM-MST(G=(V, E, w)):
    T ← {}
    for all v in V:
        key[v] ← ∞
        pi[v] ← NIL
    pick arbitrary vertex r as root; key[r]=0
    initialize priority queue Q with all v in V
    T ← {r}
    while Q is not empty:
        u ← EXTRACT-MIN(Q)
        if pi[u] != NIL:
            T ← T ∪ {(pi[u], u)}
        for each neighbour v of u:
            if v in Q and w(u, v) < key[v]:
                DECREASE-KEY(Q, v, w(u, v))
                pi[v] ← u
```

key[v] keeps the “shortest distance” between v and the current tree

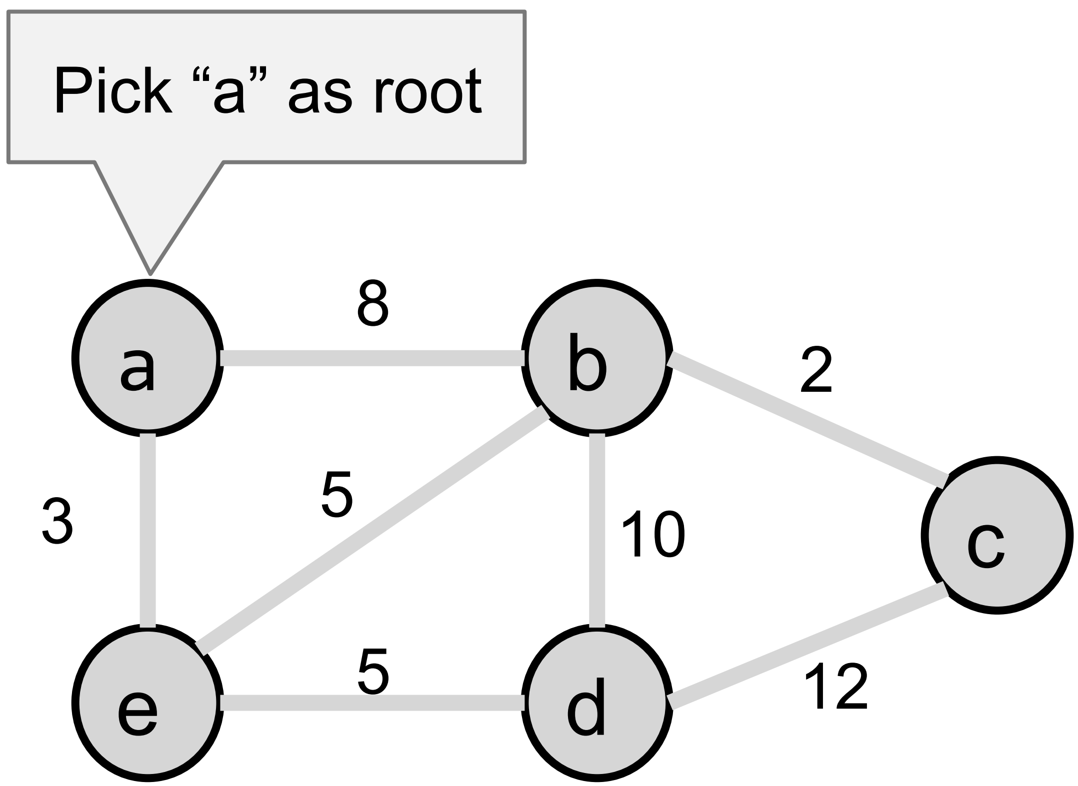
pi[v] keeps who, in the tree, is v connected to via lightest edge.

u is the next vertex to add to current tree

add edge, pi[u] is lightest vertex to connect to, “safe”

all u’s neighbours’ distances to the current tree need update

Trace an example!

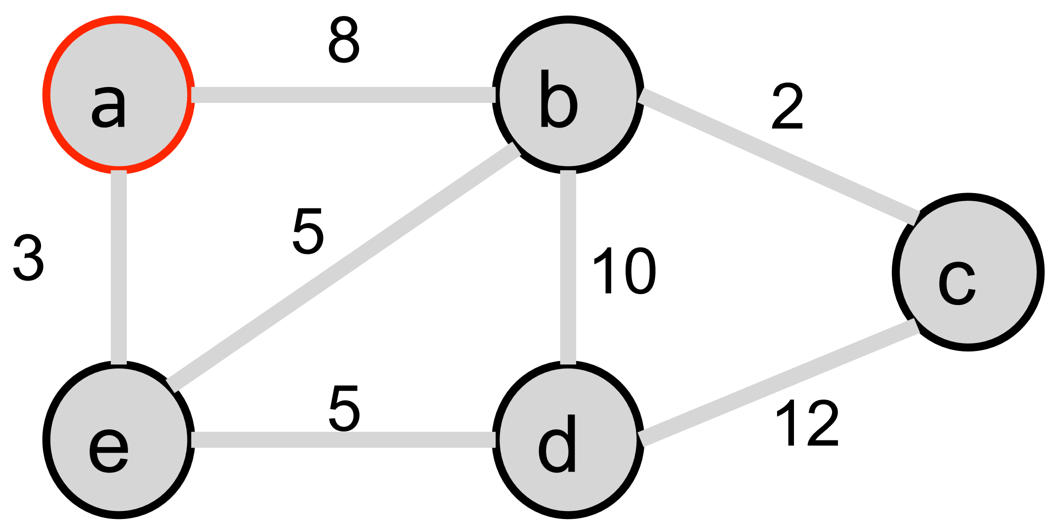


Q	key	pi
a	0	NIL
b	∞	NIL
c	∞	NIL
d	∞	NIL
e	∞	NIL

Next, ExtractMin !

ExtractMin (#1)
then update neighbours' keys

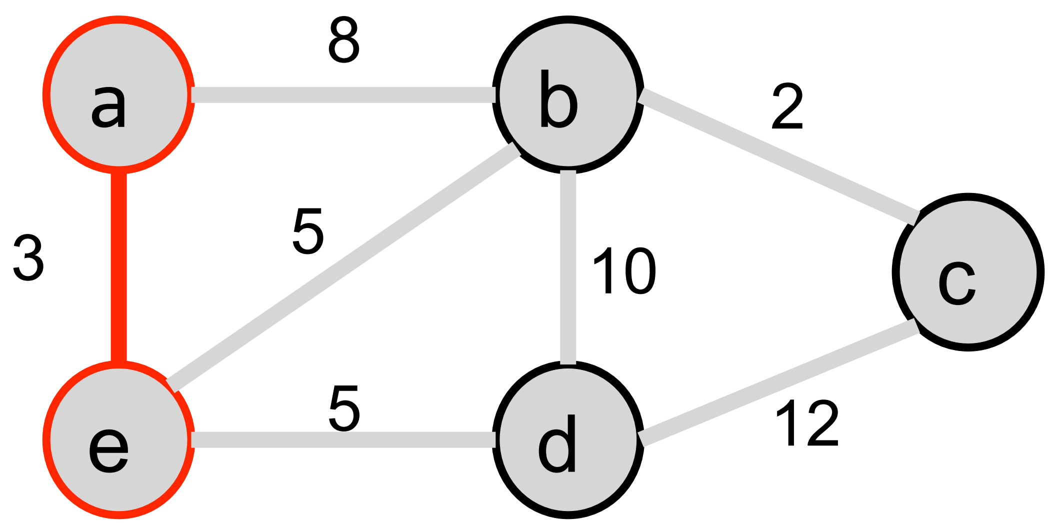
a: 0, NIL



Q	key	pi
b	∞ → **8**	NIL → **a**
c	∞	NIL
d	∞	NIL
e	∞ → **3**	NIL → **a**

ExtractMin (#2)
then update neighbours' keys

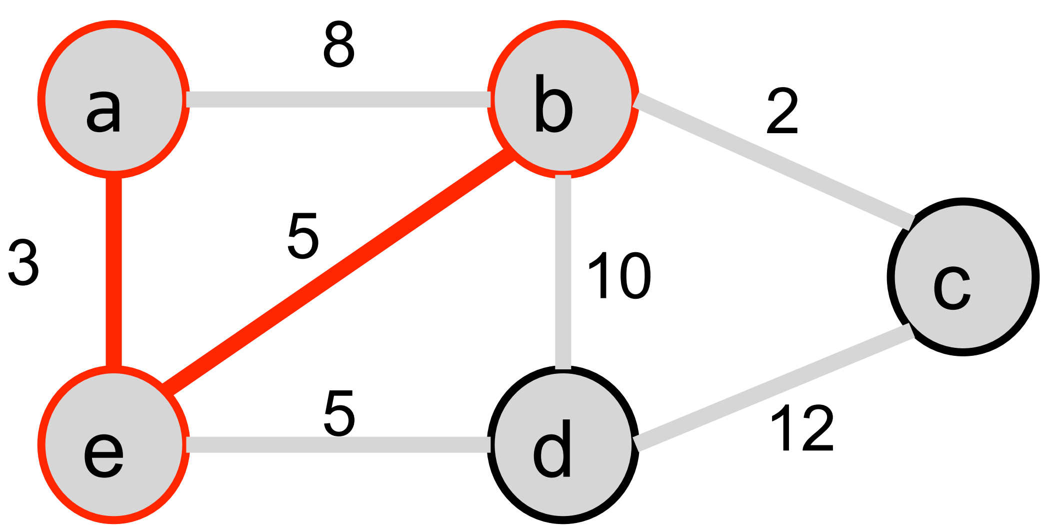
e: 3, a



Q	key	pi
b	8→**5**	a →**e**
c	∞	NIL
d	∞ →**5**	NIL →**e**

ExtractMin (#3)
then update neighbours' keys

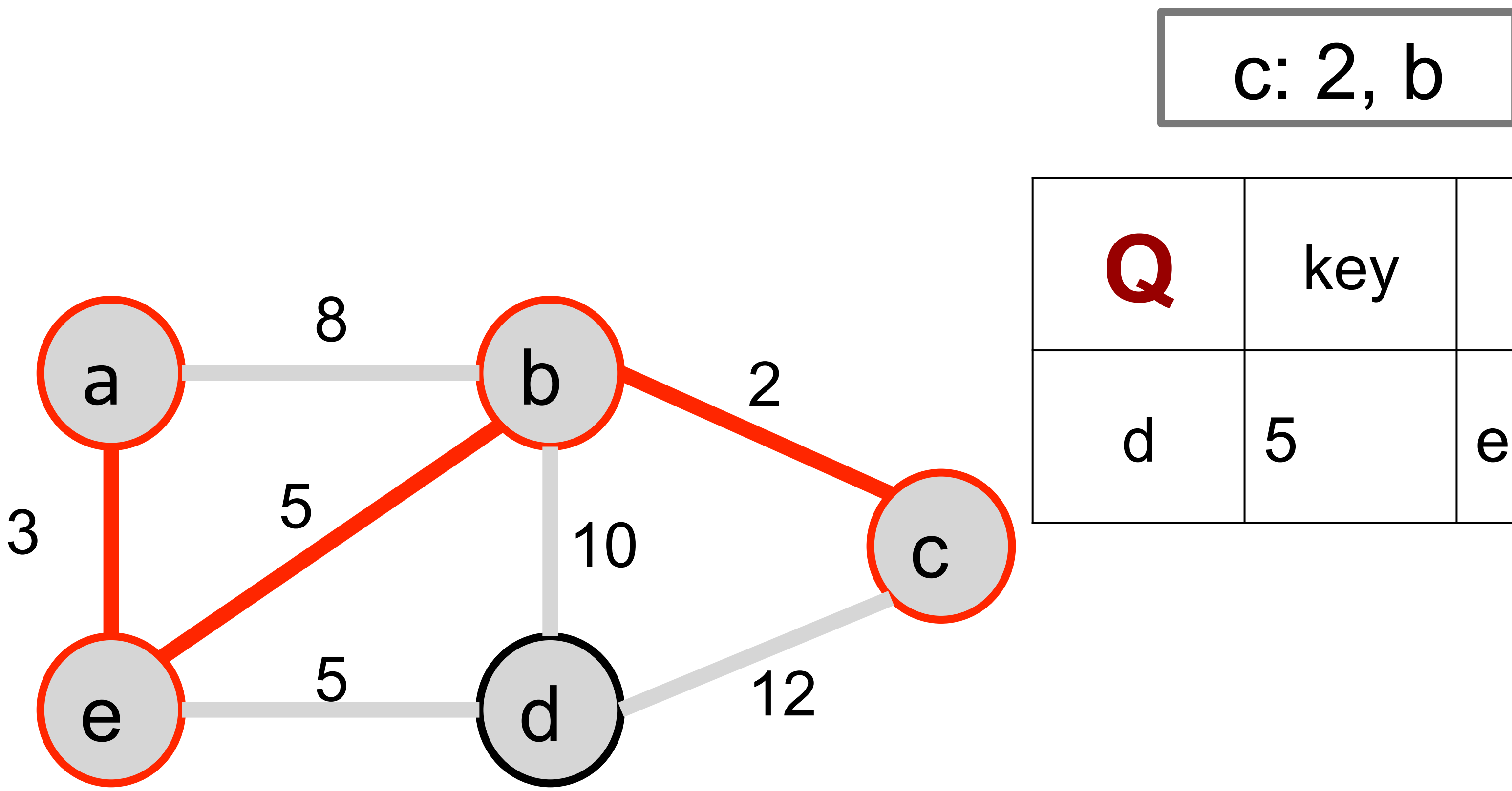
b: 5, e



Q	key	pi
c	$\infty \rightarrow$ **2**	NIL $\rightarrow$ **b**
d	5	e

Could also have extracted d since its key is also 5 (min)

ExtractMin (#4)
then update neighbours' keys



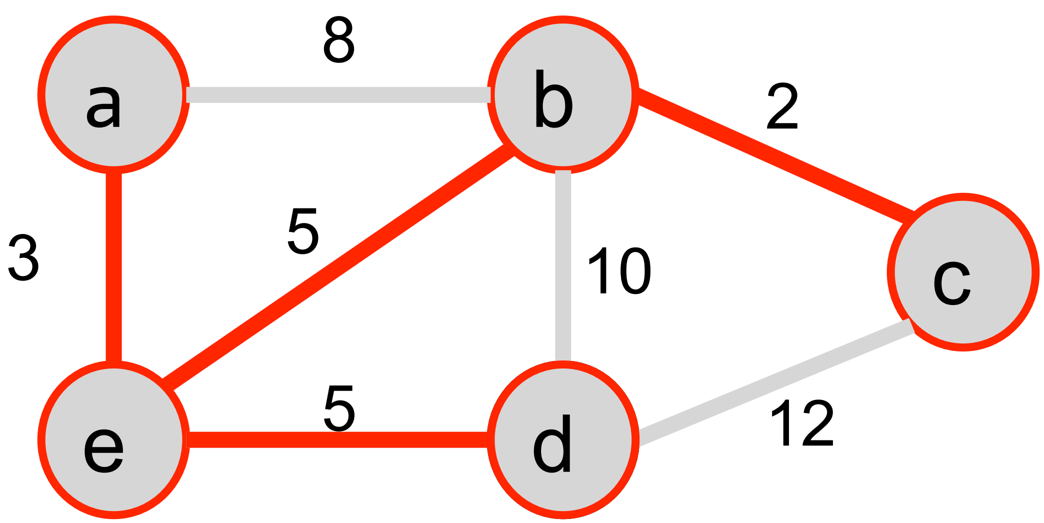
Q	key	pi
d	5	e

ExtractMin (#4)
then update neighbours' keys

d: 5, e

Q	key	pi

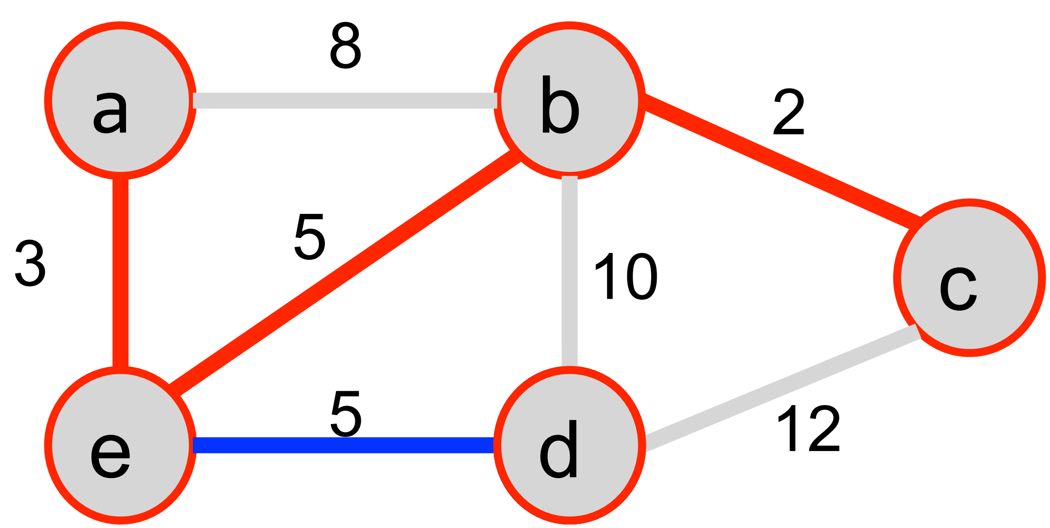
Q is empty now.



MST grown!

Correctness of Prim's

The added edge is always a "**safe**" edge, i.e., the **minimum** weight edge crossing the cut (because of **ExtractMin**).



Runtime analysis: Prim's

➔Assume we use **binary min heap** to implement the priority queue.

➔Each ExtractMin take **O(log V)**

➔In total **V** ExtractMin's

➔In total, check at most **O(E)** neighbours, each check neighbour could lead to a **DecreaseKey** which takes **O(log V)**

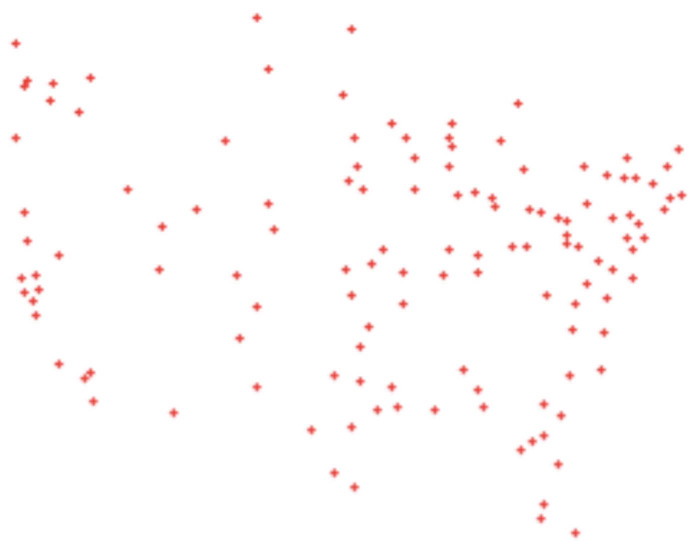
➔**TOTAL: O((V+E)log V) = O(E log V)**

In a connected graph $G = (V, E)$

$|V|$ is in $\mathbf{O(|E|)}$ because…
$|E|$ has to be at least $\mathbf{|V|-1}$

Also, $\mathbf{\log |E|}$ is in $\mathbf{O(\log |V|)}$ because …
E is at most V^2,
so $\mathbf{\log E}$ is at most $\mathbf{\log V^2 = 2 \log V}$, which is in $\mathbf{O(\log V)}$

Kruskal's MST algorithm



Kruskal's algorithm: idea

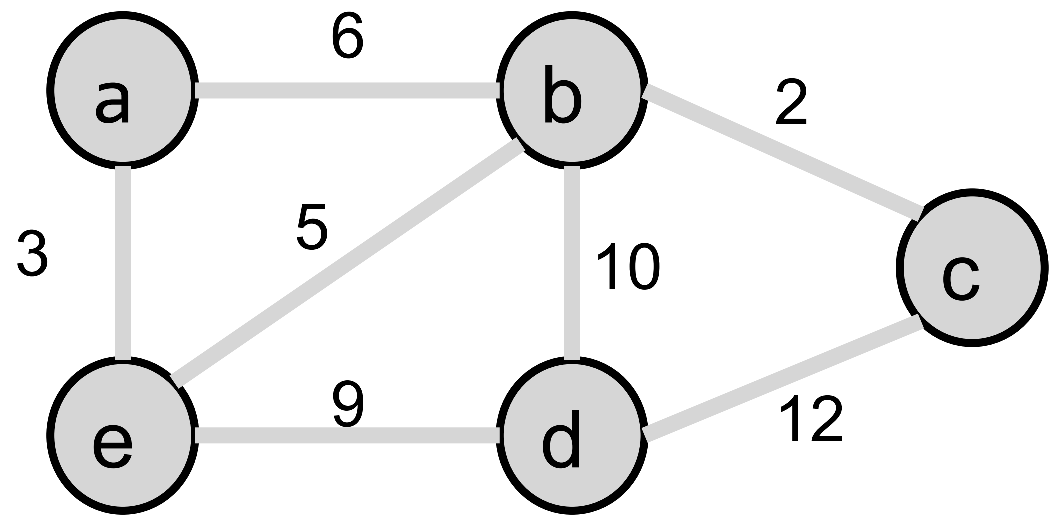
- ➔**Sort** all edges according to **weight**, then start adding to MST from the **lightest** one.
  - ◆**This is "greedy"!**
- ➔Constraint: added edge must NOT cause a **cycle**
  - ◆In other words, the two endpoints of the edge must belong to two **different** trees (components).
- ➔The whole process is like unioning small trees into a big tree.

Pseudocode

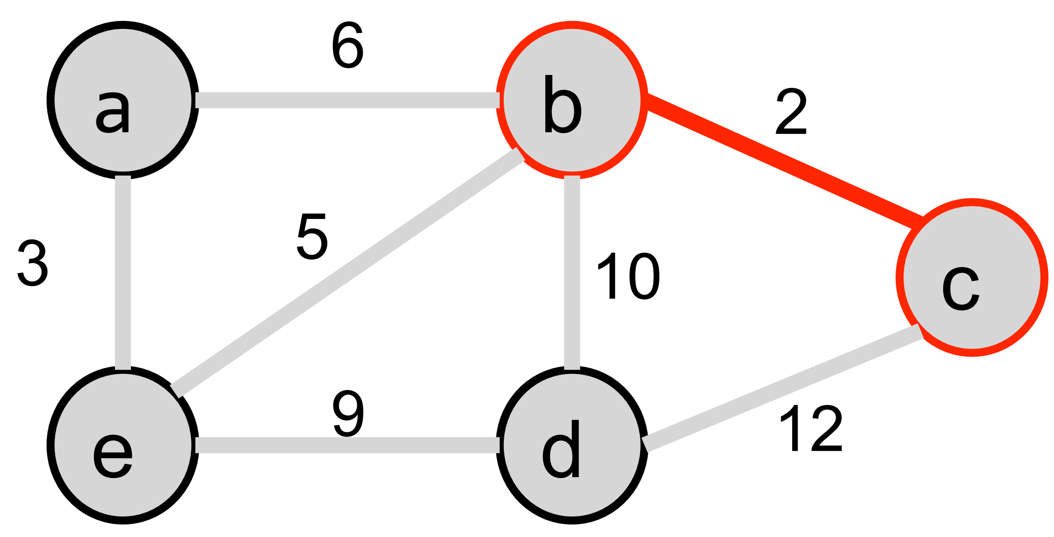
m = |E|

```
KRUSKAL-MST(G(V, E, w)):
    T ← {}
    sort edges so that w(e1)≤w(e2)≤...≤w(em)
    for i ← 1 to m:
       # let (ui, vi) = ei
       if ui and vi in different components:
          T ← T ∪ {ei}
```

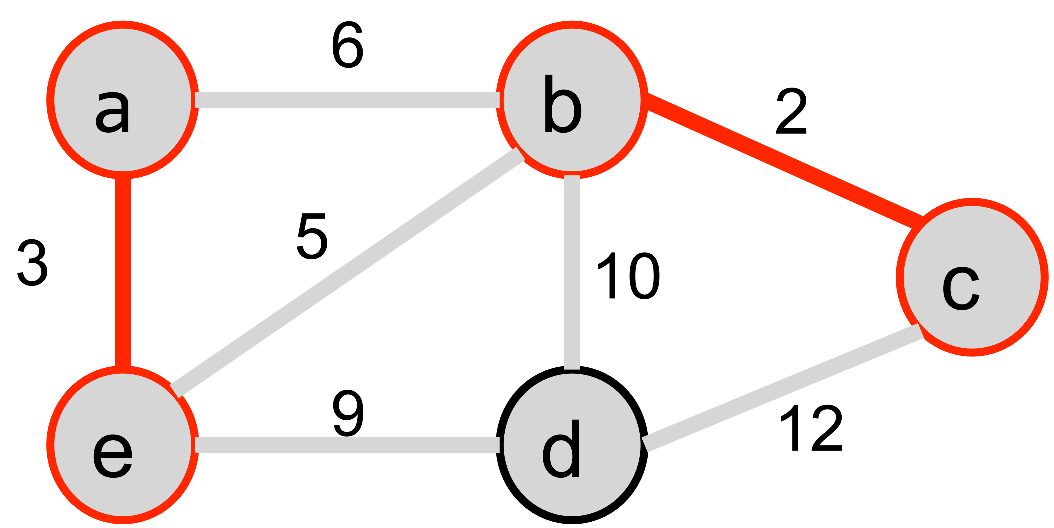
Example



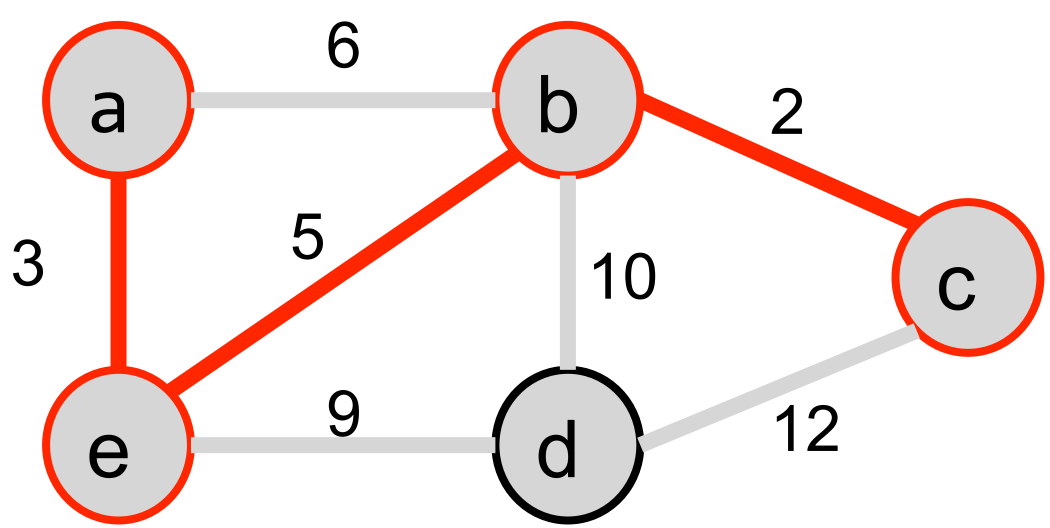
Add (b, c), the lightest edge



Add (a, e), the 2nd lightest



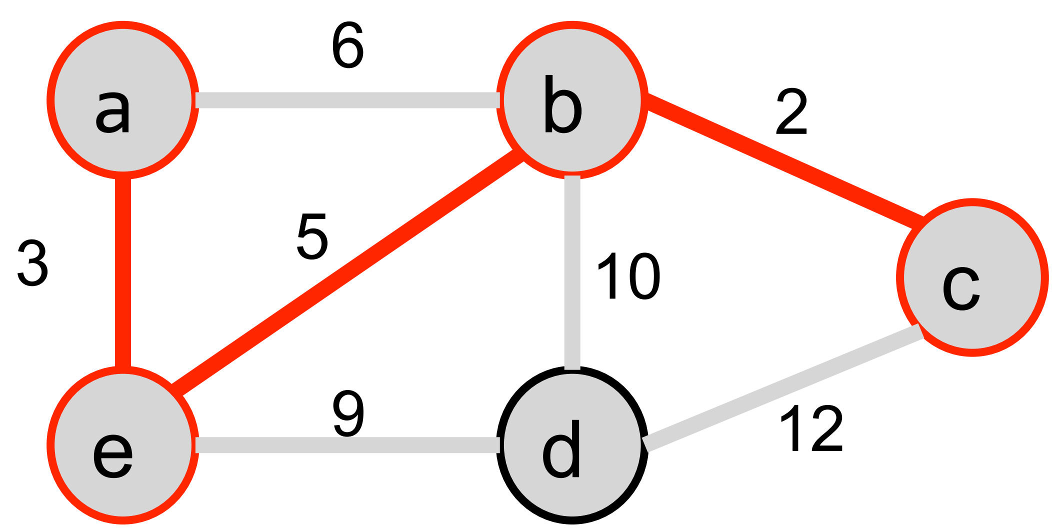
Add (b, e), the 3rd lightest



Add (a, b), the 4th lightest ...

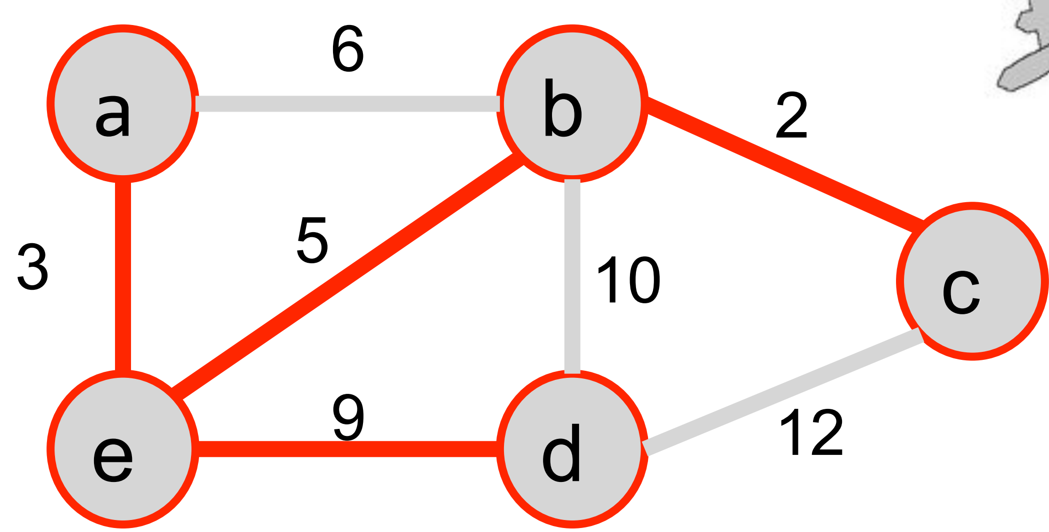
No! a, b are in the same component

Add (d, e) instead!



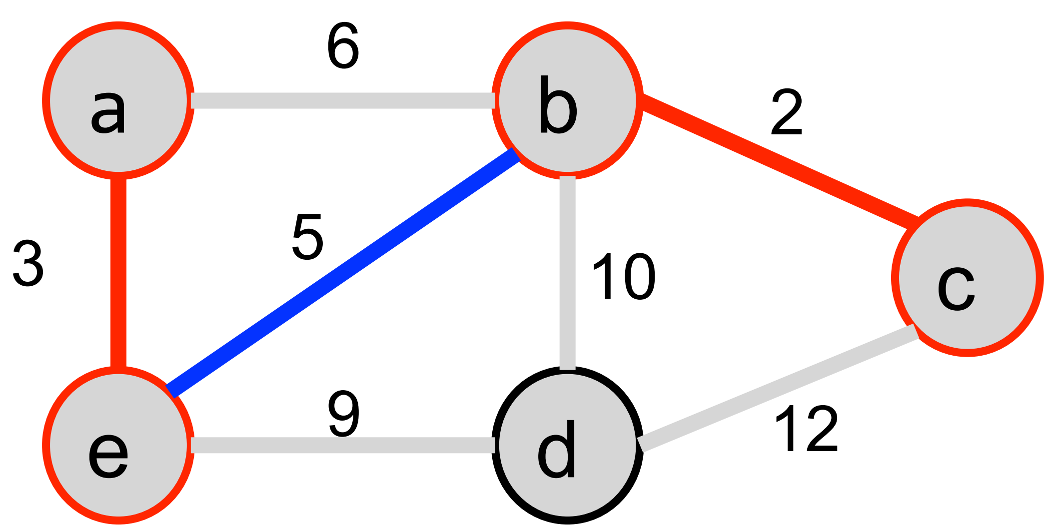
MST grown!

Add (d, e) ...



Correctness of Kruskal’s

The added edge is always a “**safe**” edge, because it is the **minimum** weight edge among all edges that **cross** components



Runtime ...

m = |E|

sorting takes **O(E log E)**

```
KRUSKAL-MST(G(V, E, w)):
    T ← {}
    sort edges so that w(e1)≤w(e2)≤...≤w(em)
    for i ← 1 to m:
        # let (ui, vi) = ei
        if ui and vi in different components:
            T ← T ∪ {ei}
```

How **exactly** do we do these two lines?

We need the Disjoint Set ADT

which stores a collections of nonempty disjoint sets **S1, S2, …, Sk**, each has a "representative".

and supports the following operations

- ➔**MakeSet(x)**: create a new set {x}
- ➔**FindSet(x)**: return the representative of the set that x belongs to
- ➔**Union(x, y)**: union the two sets that contain x and y, if different

Real Pseudocode

m = |E|

```
KRUSKAL-MST(G(V, E, w)):
    T ← {}
    sort edges so that w(e1)≤w(e2)≤...≤w(em)
    for each v in V:
       MakeSet(v)
    for i ← 1 to m:
       # let (ui, vi) = ei
       if FindSet(ui) != FindSet(vi):
          Union(ui, vi)
          T ← T ∪ {ei}
```

Next week

➔More on Disjoint Set