CSC263 Winter 2020

Graphs: MST

Lecture 9

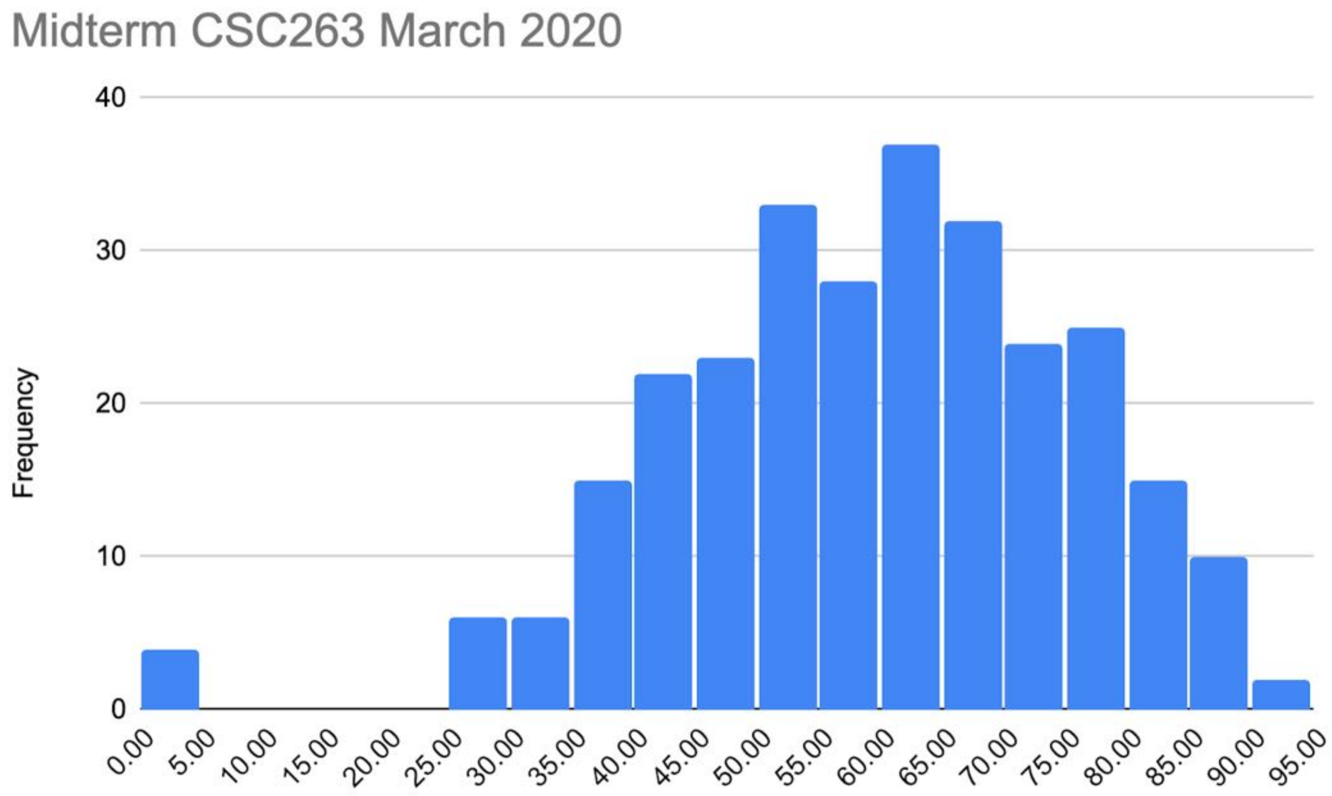
Midterm Results

Average 32/54 (60%)

Median 32.5/54

Highest Mark 49.5/54

in this Section



Midterm Remarking Requests

Remarking request

- Check solutions posted on the forum
- Fill in the remarking request form (posted on the course website)
- Staple it to your test and **submit to Sushant** by end of next week.
- A subset of the tests have been scanned, so don't commit AO by altering your answer and remark.

Make sure your mark is correct on MarkUs

Observations & Reflections

- Q1-3 were essentially from lectures / tutorials. If you didn’t do well, you need to change your way of learning for this course.
- Make your mistakes worthwhile. Make sure you understand the problem/solution, and will be able to solve it next time.
- If you’re not sure how to improve, feel free to talk to Sushant or Jessica about how to improve your learning for the rest of the semester. It’s not too late yet!

Announcement

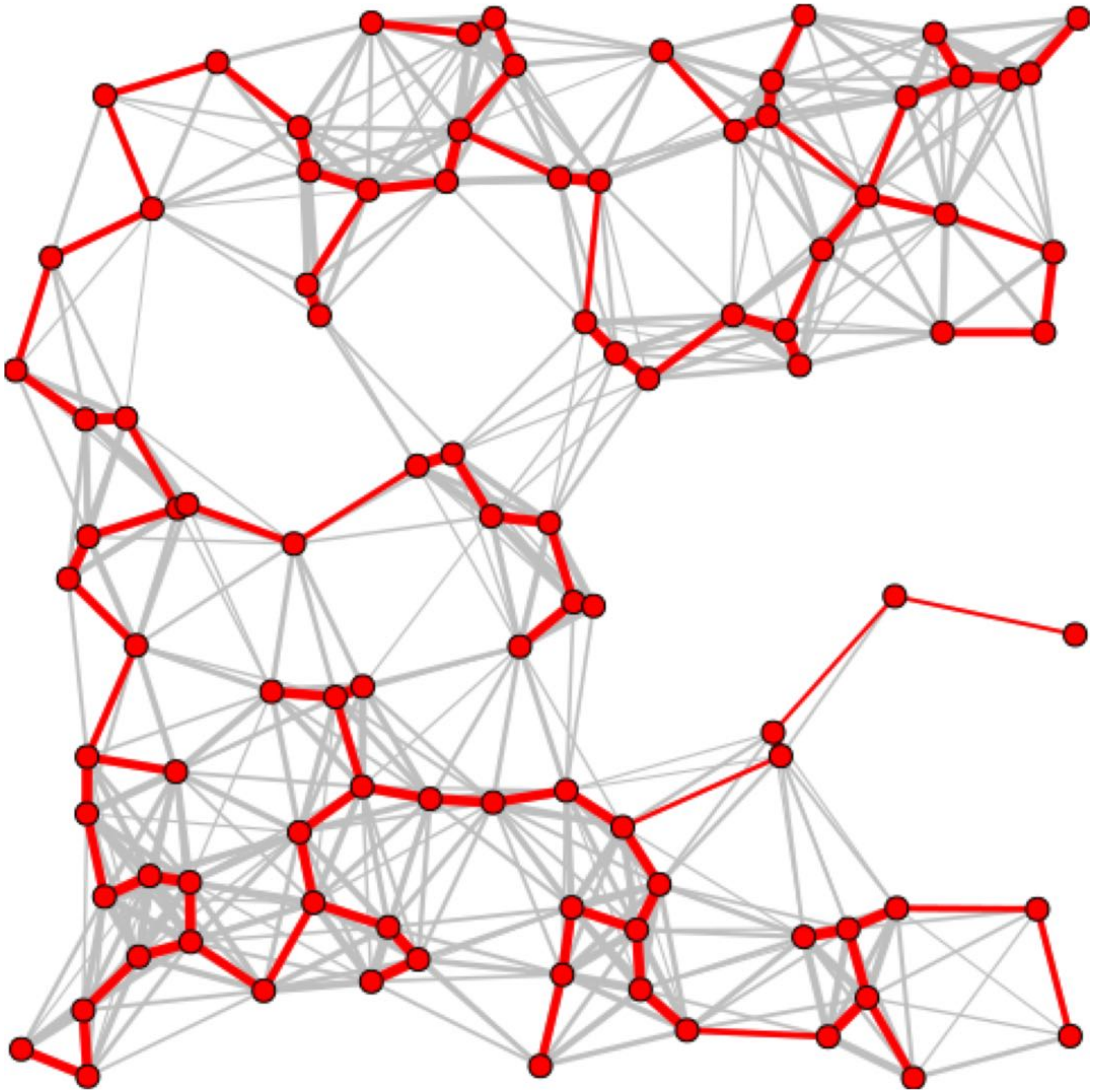
PS3 will be out by the beginning of next week!

MST

Minimum

Spanning

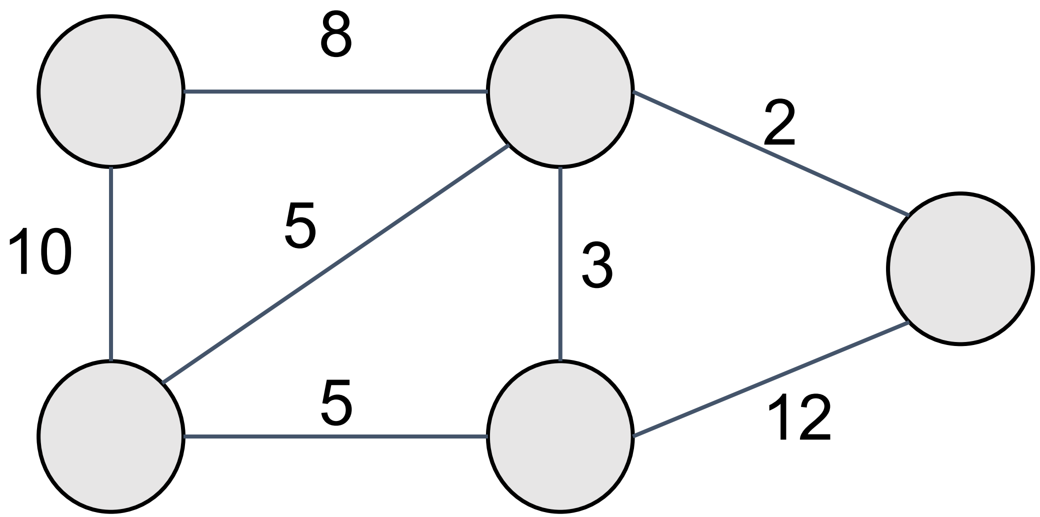
Tree



Graph of Interest Today

A **connected undirected weighted** graph

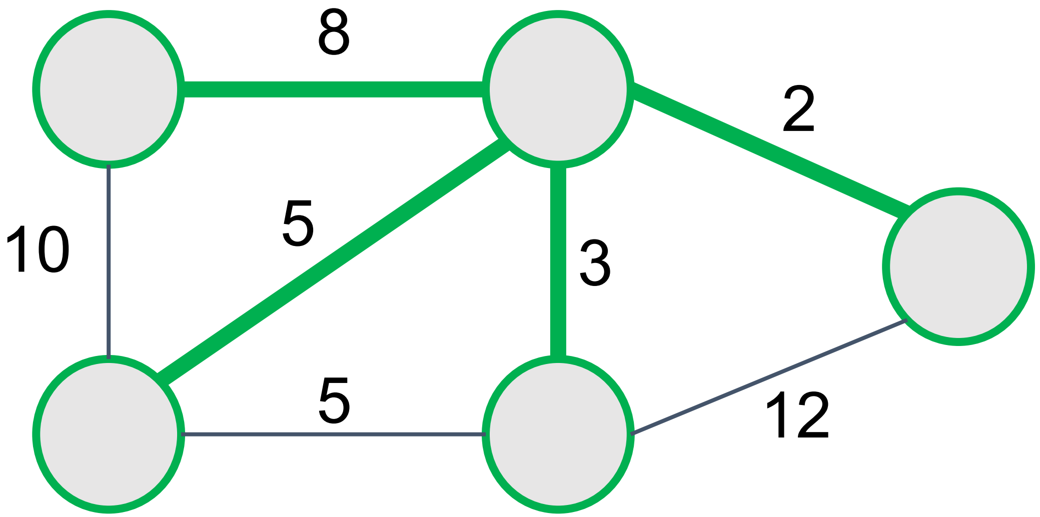
G = (V, E) with weights **w(e)** for each e ∈ E



Minimum Spanning Tree

Minimum	it has the smallest total weight
Spanning	it covers all vertices in the graph
Tree	it is a connected, acyclic subgraph

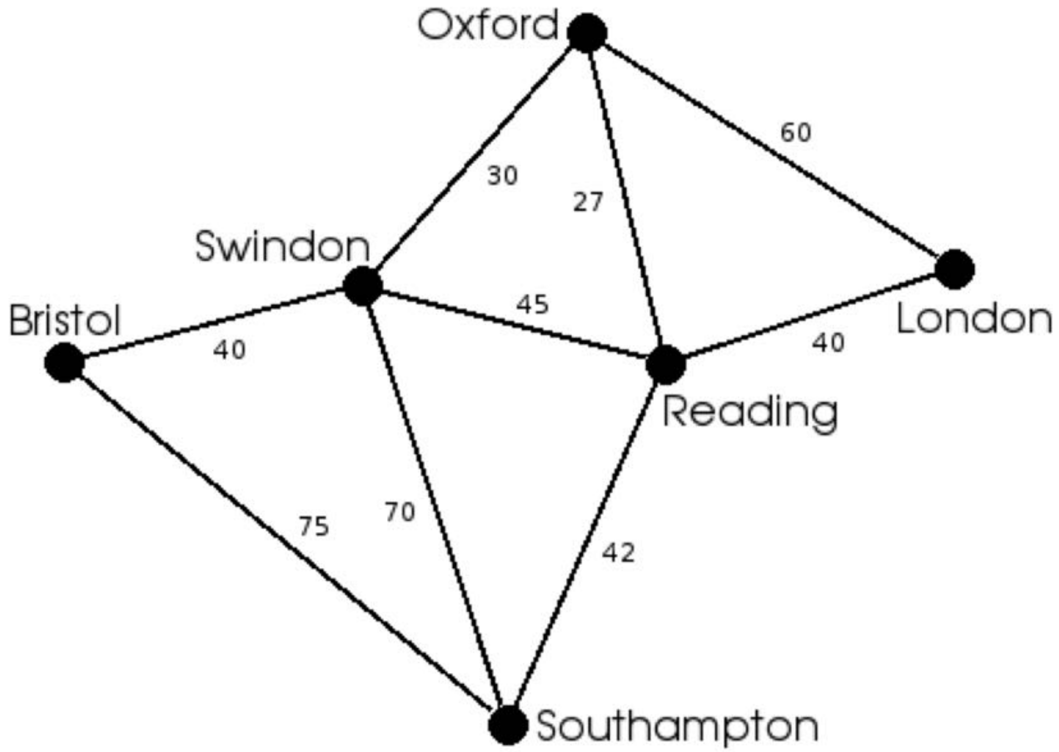
A Minimum Spanning Tree



may not be unique…

Applications of MST

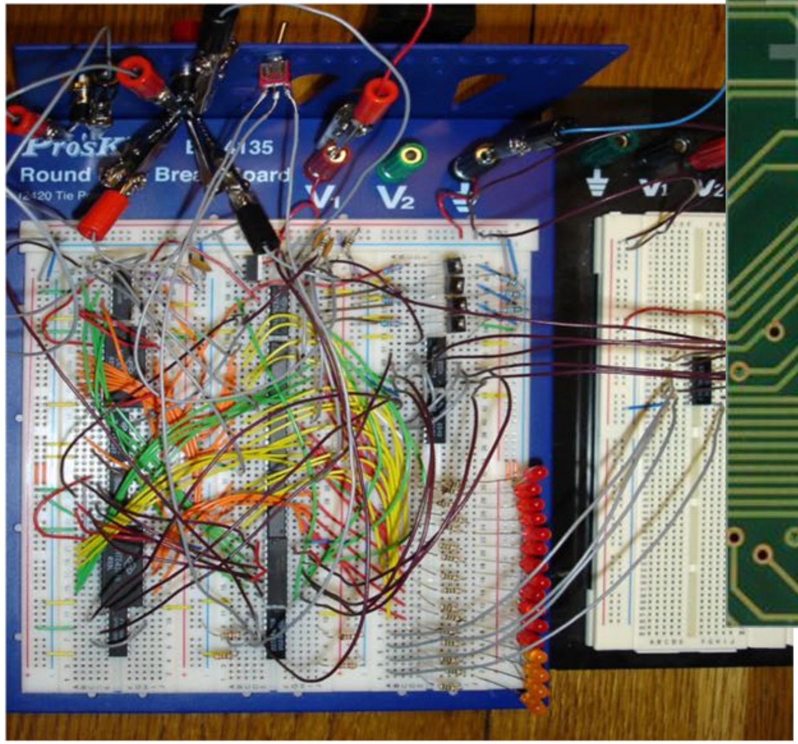
Build a broadband network that connects all towns and with the minimum cost.

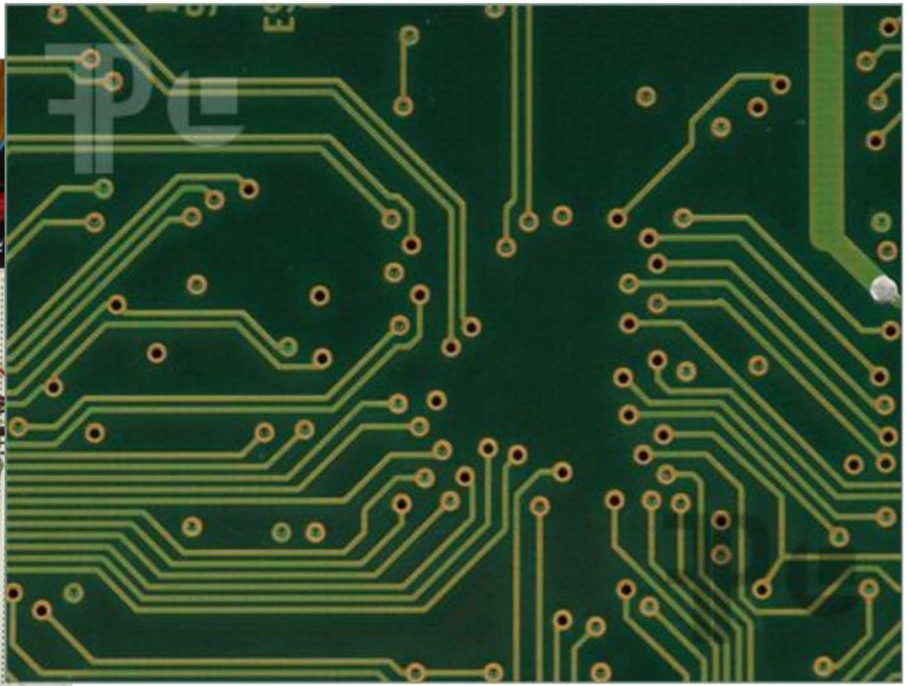


Applications of MST

Circuit/network Design

Connect all components with the least amount of wiring.





Other applications

Cluster Analysis

Approximation Algorithms for hard problems

- e.g. Traveling Salesman

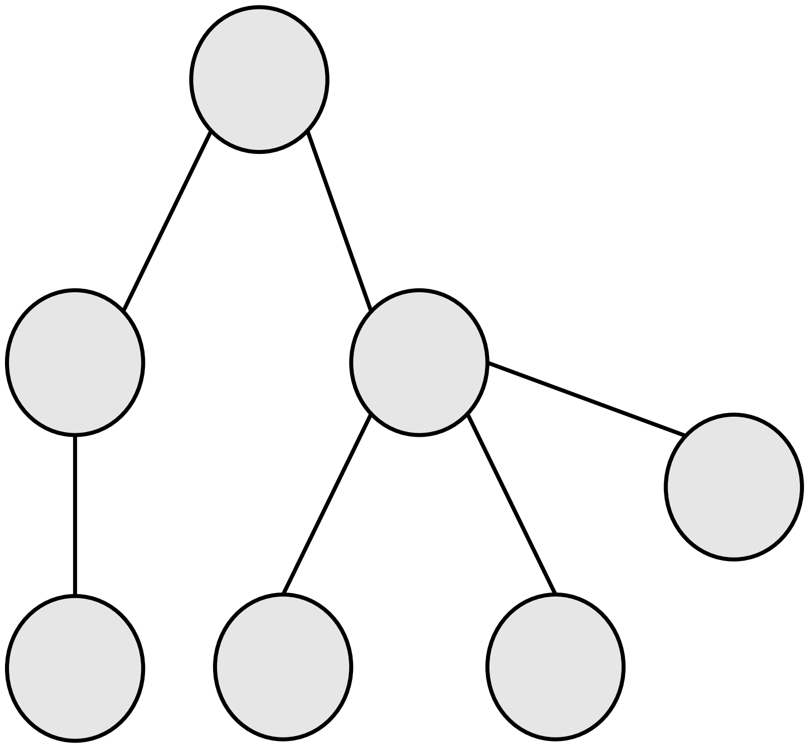
In order to understand

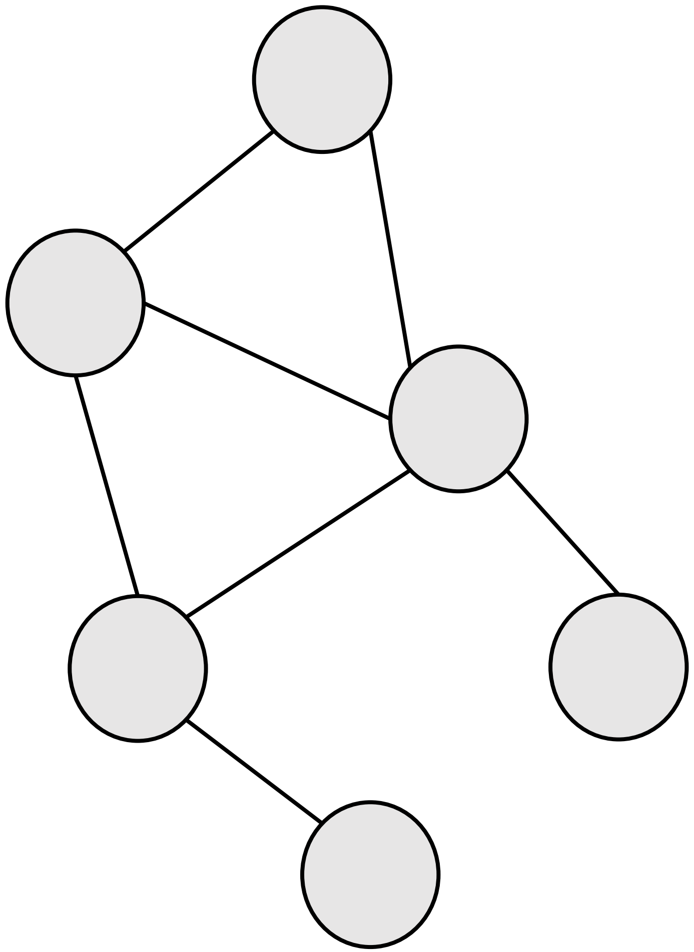
minimum spanning tree

we need to first understand

tree

Tree is a special type of **Graph**





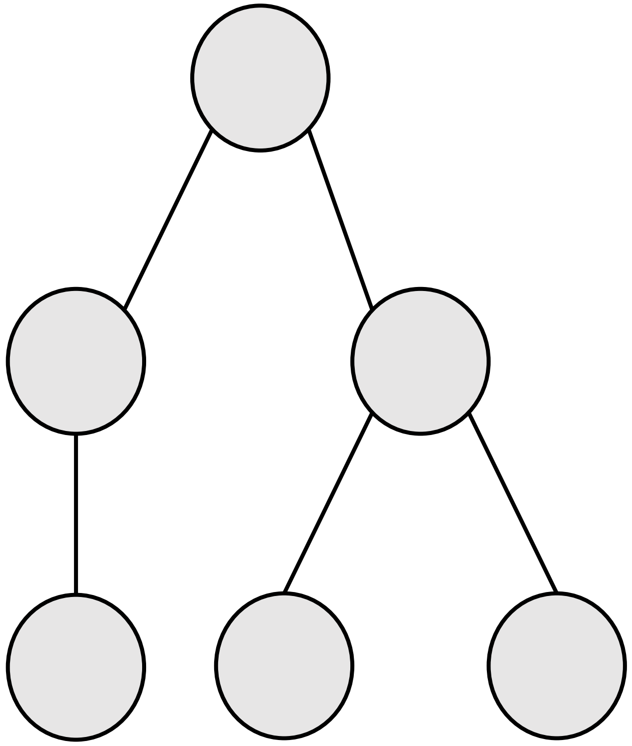
Tree

A tree is a 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__.



MST Properties

The MST of a connected graph G=(V,E) has

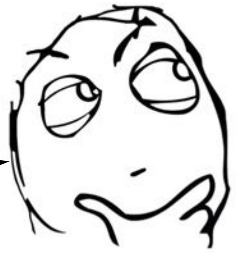
- |V| vertices (because spanning)
- |V| - 1 edges (because tree)

MST Algorithms

Idea #1

Start with **T = G.E** and remove edges until an MST remains.

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



Idea #2

Start with **empty T**, and add edges until an MST is built.

Hint

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

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

Idea #1

Note: Here T is an edge set

Start with **T = G.E** and remove edges until an MST remains.

Worst Case

We have to remove $\binom{|V|}{2} - (|V| - 1) = O(|V|^2)$ edges.

Idea #2

Start with **empty T**, and add edges until an MST is built.

Worst Case

We have to add only $O(|V|)$ edges.

➢ MST algorithms that add edges are more efficient than those removing edges.

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 ← Ø
    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” Edge e for T

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

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

<u>Assumption</u>

Before adding e, **T ⊆ some MST.**

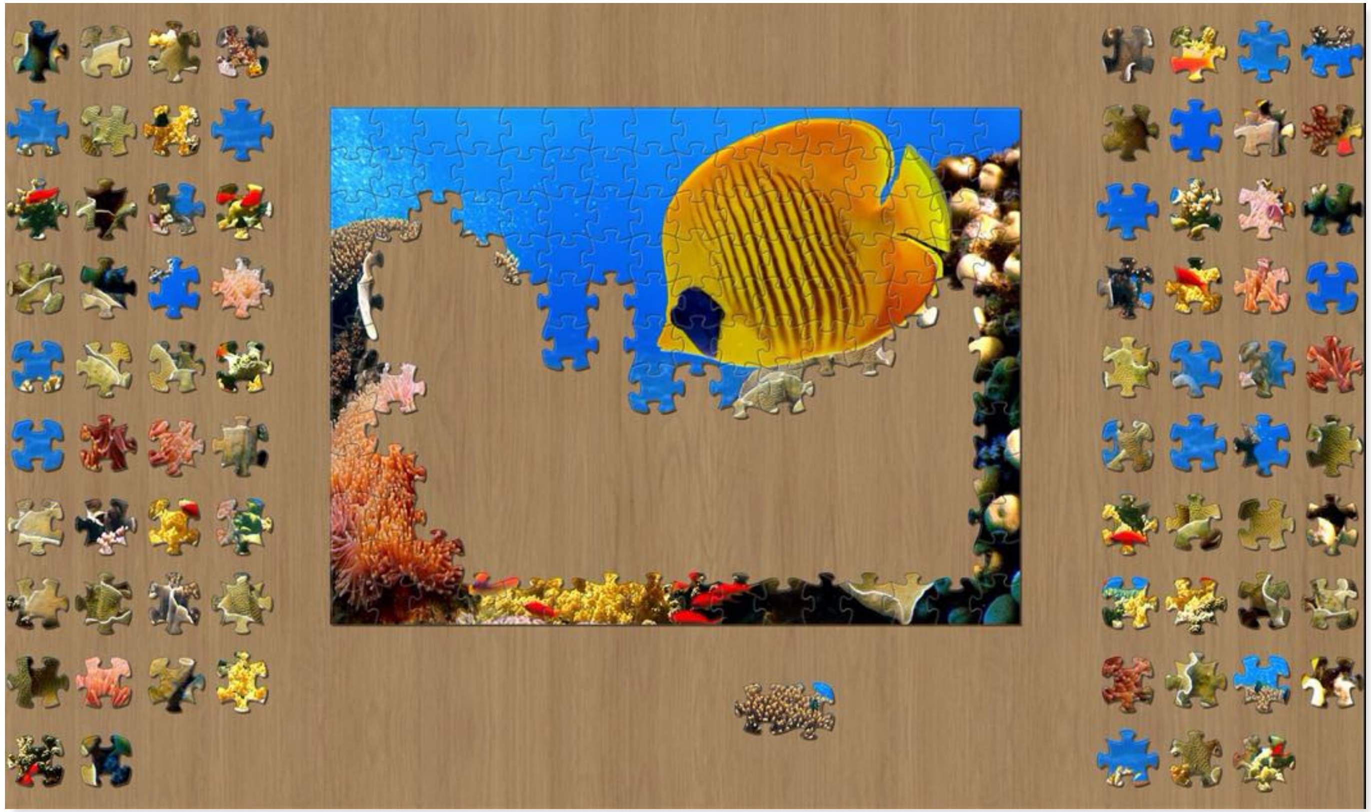
Edge **e** is safe if **after** adding **e**, still **T ⊆ some MST**

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

(Easily proven by induction)

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

Prim's Algorithm

Kruskal's Algorithm

They are both based on one Theorem...

The Theorem

CLRS Theorem 23.1

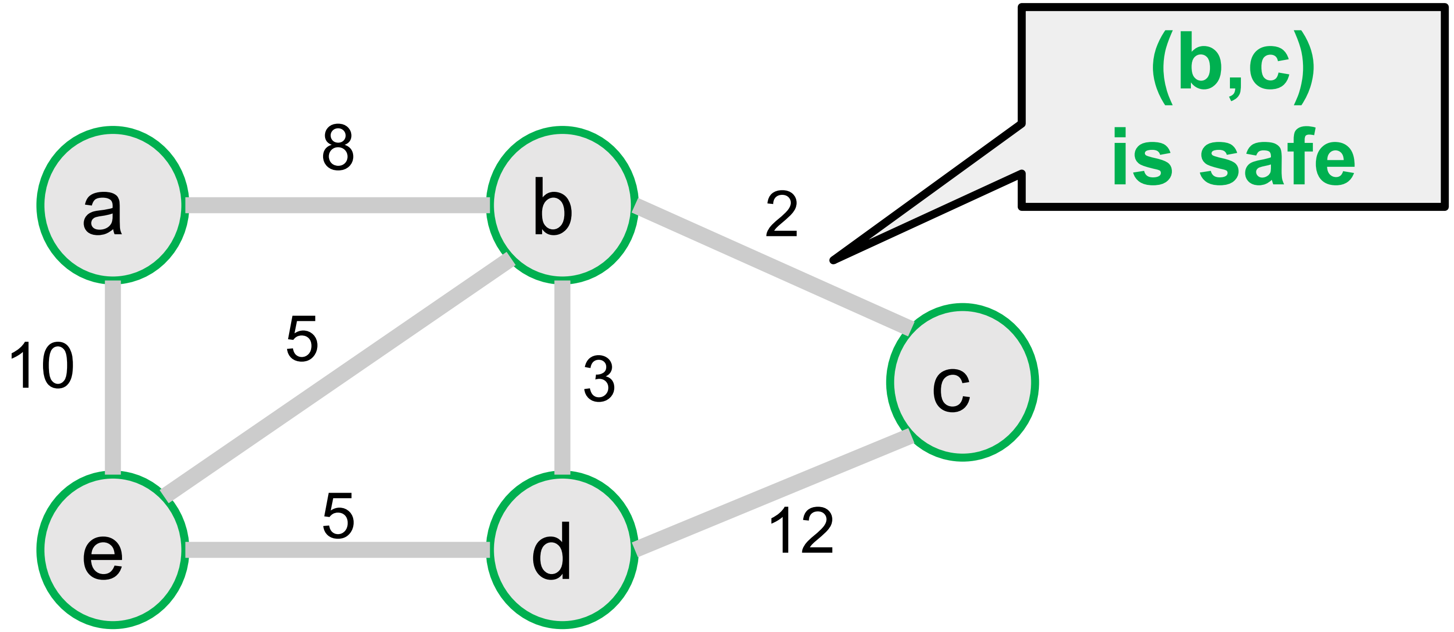
Let **G** be a connected, undirected, weighted graph, and **T** be a **subgraph** of G which is a **subset** of some MST of **G**.

Let edge **e** be the **minimum** weighted edge among all edges that **leave** a fixed **connected component** of T.

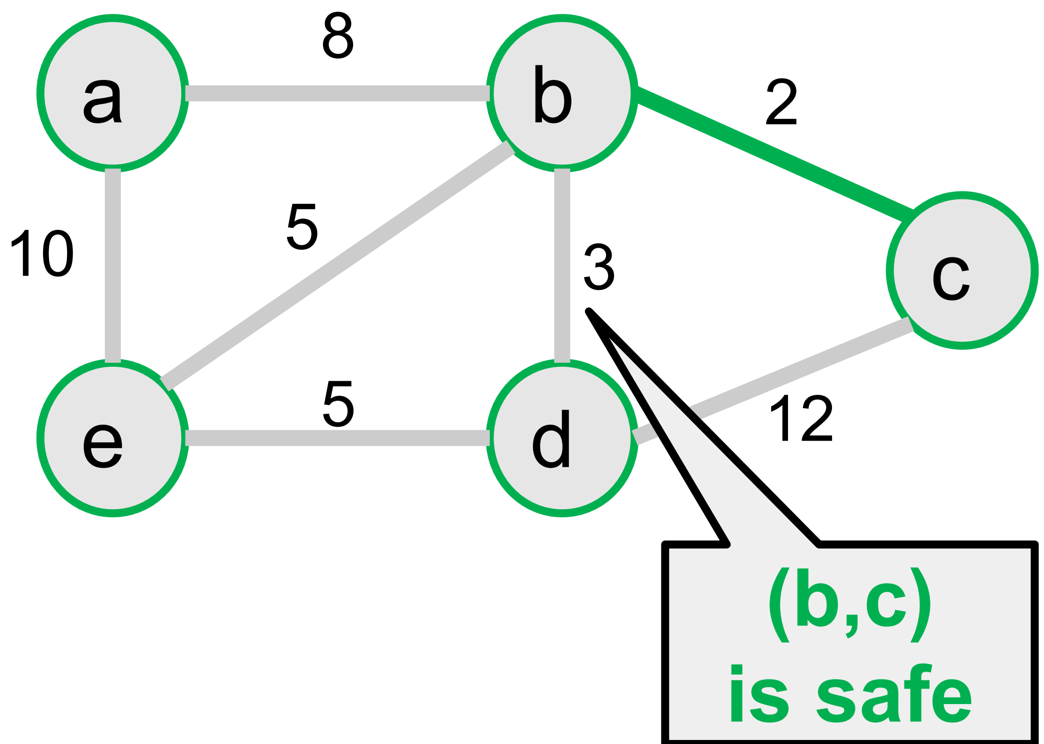
Then **e** is **safe** for **T**.

Note: Here T includes both vertices and edges

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

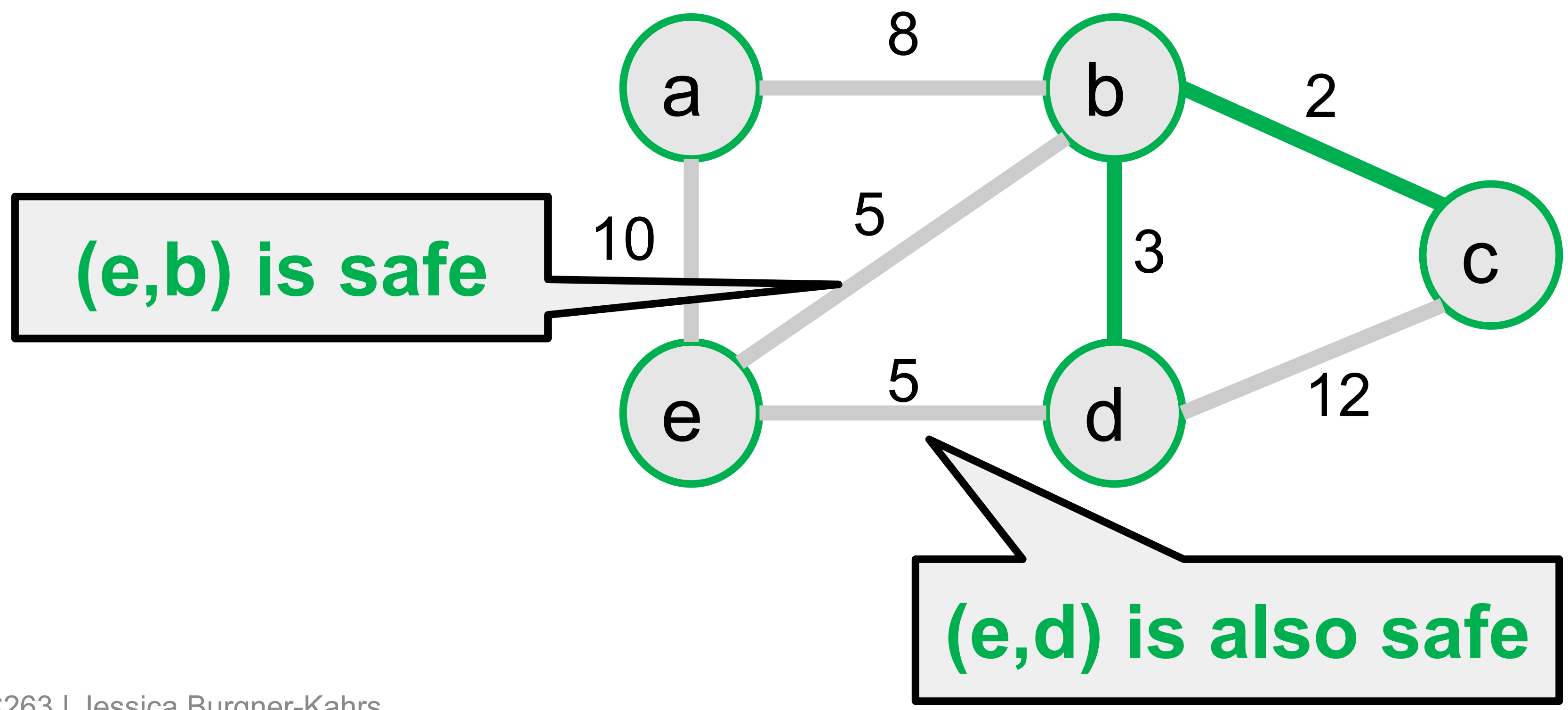


Now **b and c** in one connected component,
each of the other vertices is a component, i.e.,
4 components {b,c},{a},{d},{e}.
All gray edges are crossing components.



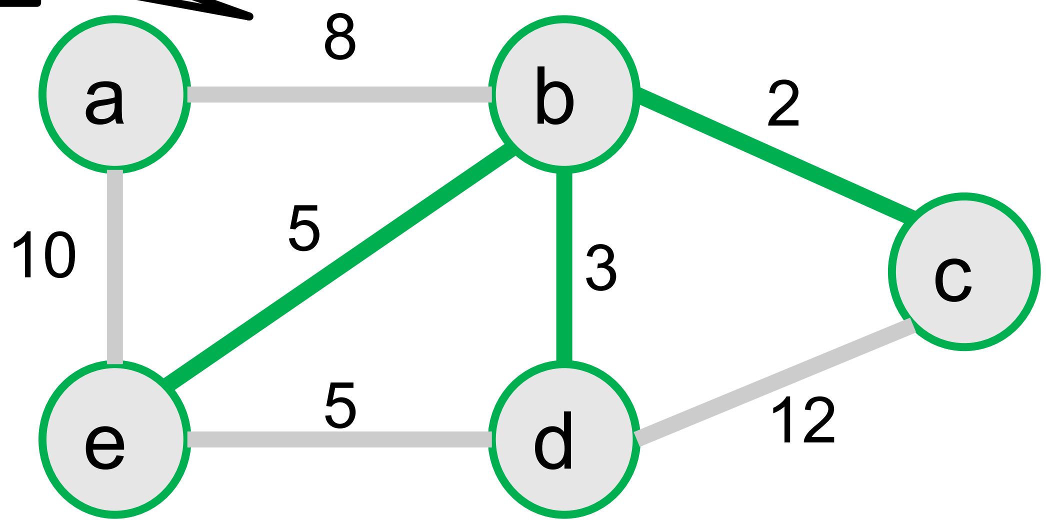
Now **b, c and d** are in one connected component, **a** and **e** each is a component, i.e. 3 components {b,c,d},{a},{e}

(c, d) is **NOT** crossing components!

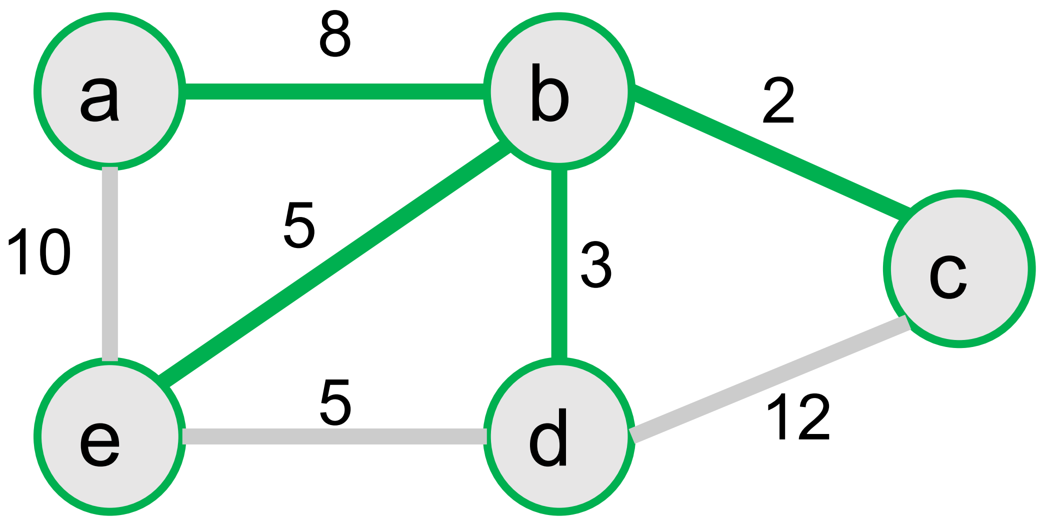


Now **b, c, d and e** are in one connected component, **a** is a component, i.e.
2 components {b,c,d,e},{a}
(a, e) and **(a, b)** are crossing components.

(b,a) is safe



MST grown!



Two Implementation Challenges

1. How to keep track of the **connected components**?
2. How to efficiently find the **minimum** weighted edge?

Prim's and **Kruskal's** basically use

different **data structures** to do these two things.

Overview: Prim's and Kruskal's

	Keep track of connected components	Find safe edge
Prim's	*one tree plus isolated vertices*	*Priority Queue ADT*
Kruskal's	*Disjoint Set ADT*	*Sort all edges by weight*

7 min

Prim’s MST algorithm

Prim’s Algorithm: Idea

- Start from an arbitrary vertex as root
- Focus on growing **one** tree, add one edge at a time. The tree is one **component**, each of the other (**isolated**) vertices is a **component**.
- Add which edge? Among all edges that are **leave the current tree** (**edges crossing components**), pick one with the **minimum** weight.
- 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
    Initialize priority queue Q with all v in V
    Pick arbitrary vertex r as root
    key[r] ← 0
    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
```

u is the end point of the “safe” edge leaving the current tree

add u to the tree using its safe edge

```
PRIM-MST(G=(V, E, w)):
    T ← {}
    for all v in V:
        key[v] ← ∞
        pi[v] ← NIL
    Initialize priority queue Q with all v in V
    Pick arbitrary vertex r as root
    key[r] ← 0
    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

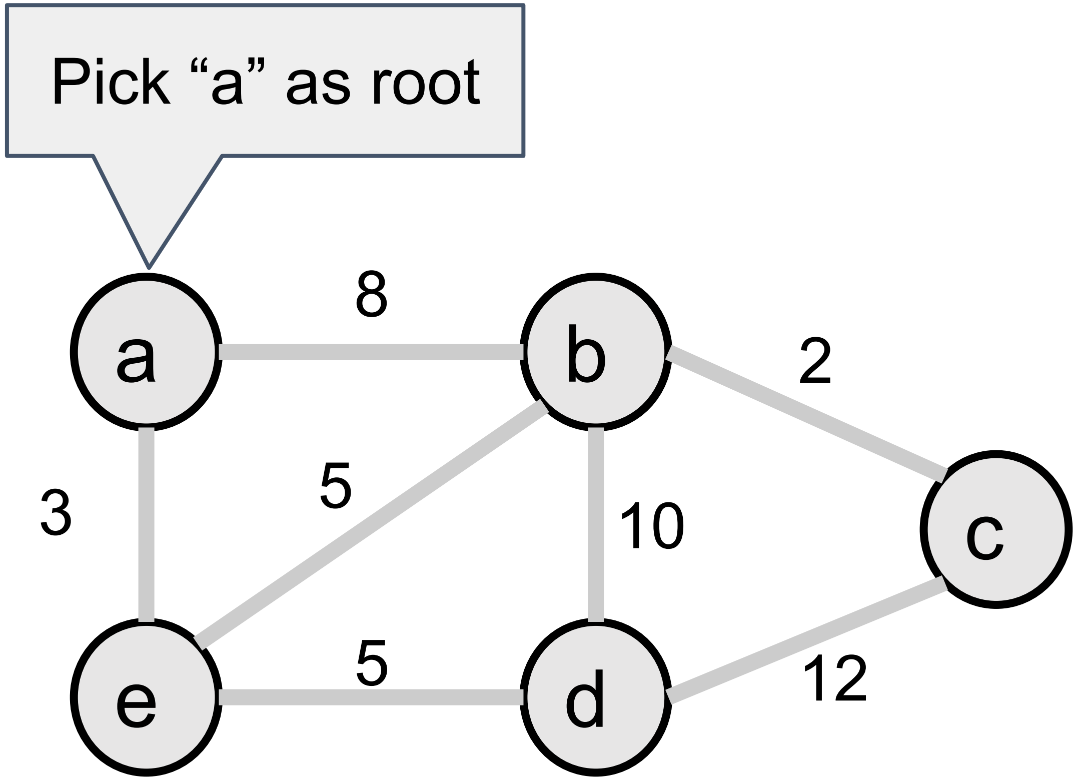
pi[v] keeps who, in the tree, is v connected to via lightest edge.

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

Example

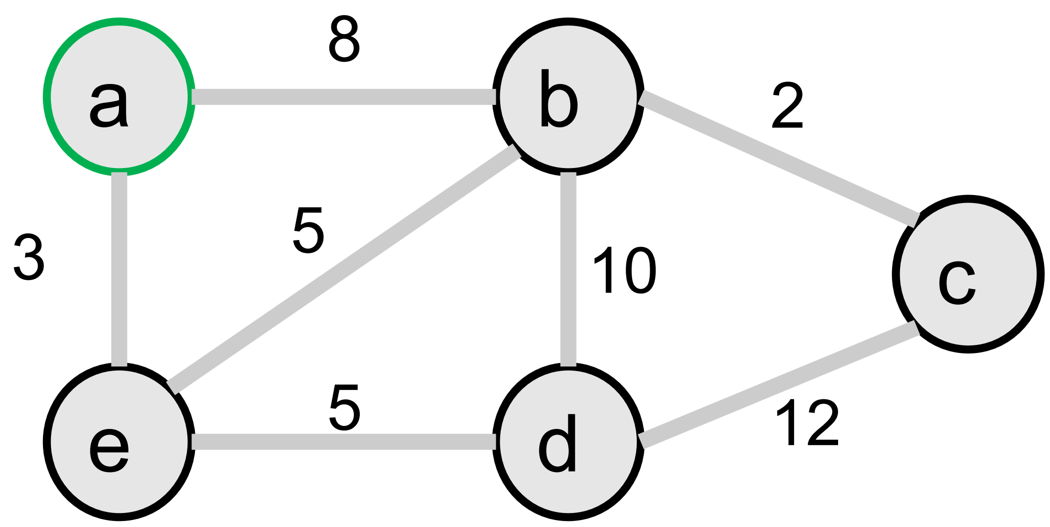


Q	key	pi
a	0	NIL
b	∞	NIL
c	∞	NIL
d	∞	NIL
e	∞	NIL

Next, ExtractMin !

ExtractMin (#1) then update neighbours' keys

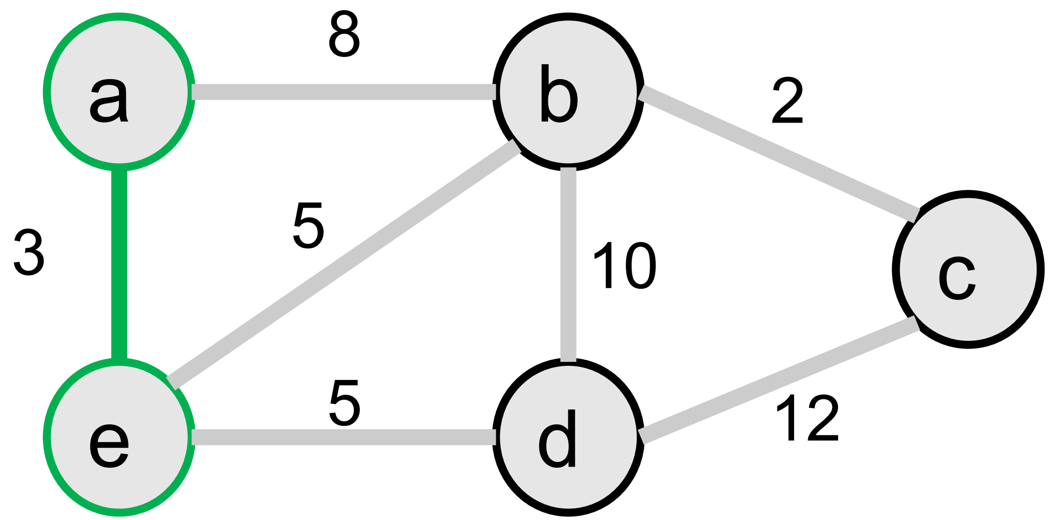
a: 0, NIL



Q	key	pi
b	$\infty \rightarrow$ **8**	NIL $\rightarrow$ **a**
c	∞	NIL
d	∞	NIL
e	$\infty \rightarrow$ **3**	NIL $\rightarrow$ **a**

ExtractMin (#2) then update neighbours' keys

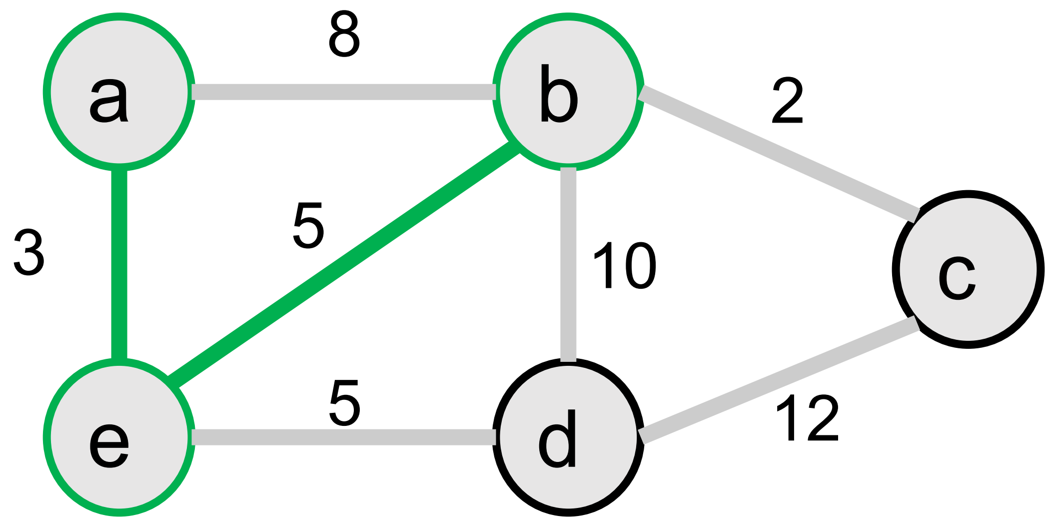
e: 3, a



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

ExtractMin (#3) then update neighbours' keys

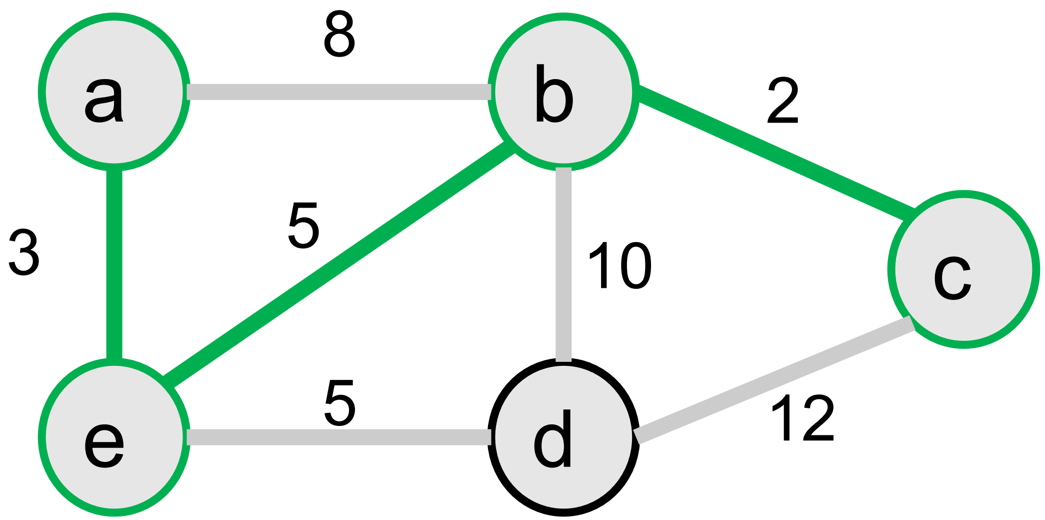
b: 5, e



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

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

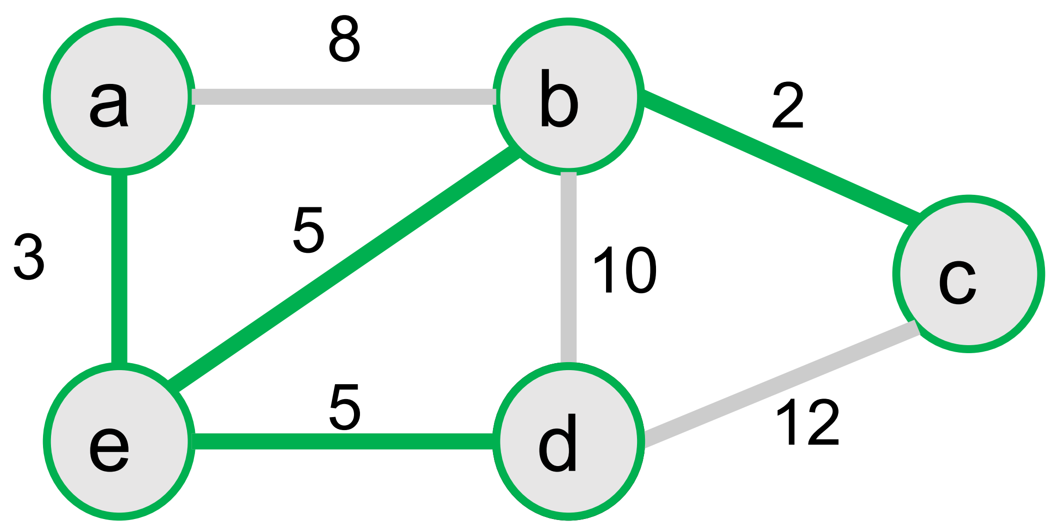
ExtractMin (#4) then update neighbours' keys



c: 2, b

Q	key	pi
d	5	e

ExtractMin (#4)
then update neighbours' keys



d: 5, e

Q	key	pi

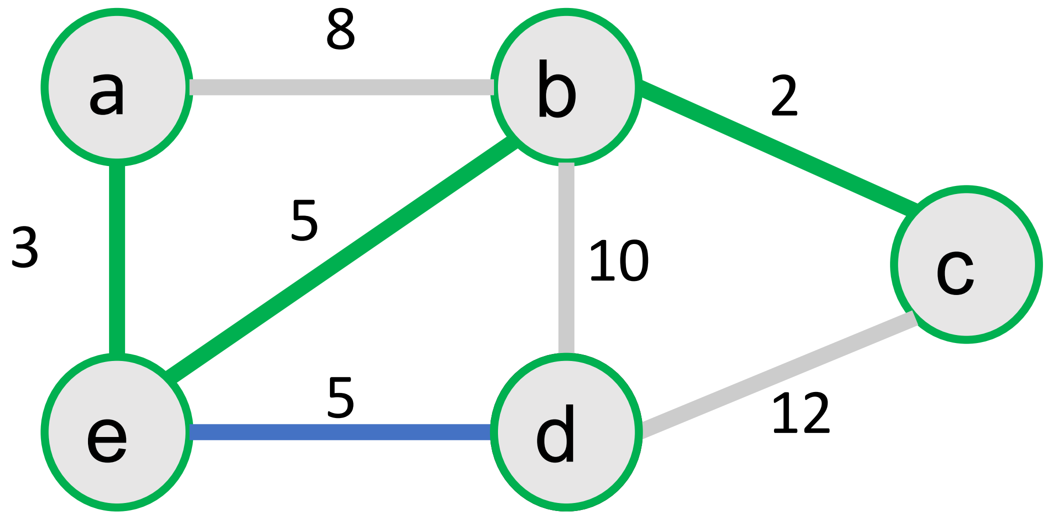
Q is empty now.

MST grown!

Correctness of Prim's

CLRS Theorem 23.1

The added edge is always a "**safe**" edge, i.e., the **minimum** weight edge leaving the current tree (because **ExtractMin**).



Runtime Analysis: Prim's

- Assume we use **binary min heap** to implement priority queue.
  - Each `ExtractMin` takes **O(lg |V|)**
  - In total **O(|V|)** `ExtractMins`
  - Total for all `ExtractMin` calls **O(|V| lg |V|)**

Runtime Analysis: Prim's

Total so far: O(|V| lg |V|)

- We look at each of the **|E|** edges once
- Worst case: Each leads to a `DecreaseKey`
- `DecreaseKey` costs **O(lg |V|)** time
- Total work for all `DecreaseKeys` is **O(|E| lg |V|)**

Runtime Analysis: Prim's

Total: $O(|V| \lg |V|) + O(|E| \lg |V|)$

$= O((|V|+|E|)\log |V|)$

This is $O(|E| \log |V|)$ in a connected graph.

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

$|V|$ is in **$O(|E|)$** because…

$|E|$ has to be at least **$|V|-1$**

Also, **$\log |E|$** is in **$O(\log |V|)$** because …

E is at most V^2,

so **$\log E$** is at most **$\log V^2 = 2 \log V$,** which is in **$O(\log V)$**

Kruskal’s MST algorithm

Kruskal’s Algorithm: Idea

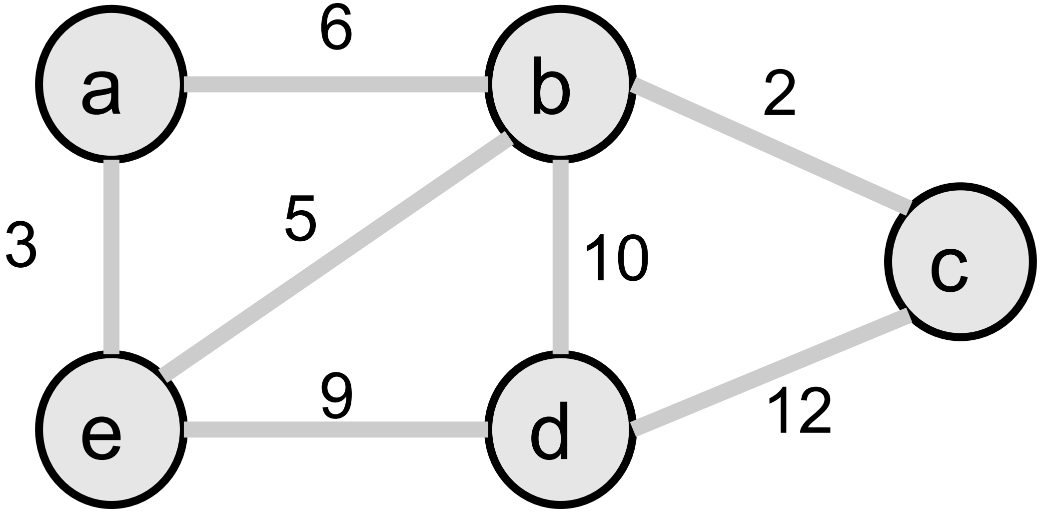
- Sort all edges according to weight, then start adding to MST from the lightest one.
- Constraint: Added edge must **NOT cause a cycle**
  - In other words, the two endpoints of the edge must belong to two different trees (components).
- Unlike Prim, Kruskal allows multiple tree components to exist and progressively combines them into larger trees

Pseudocode

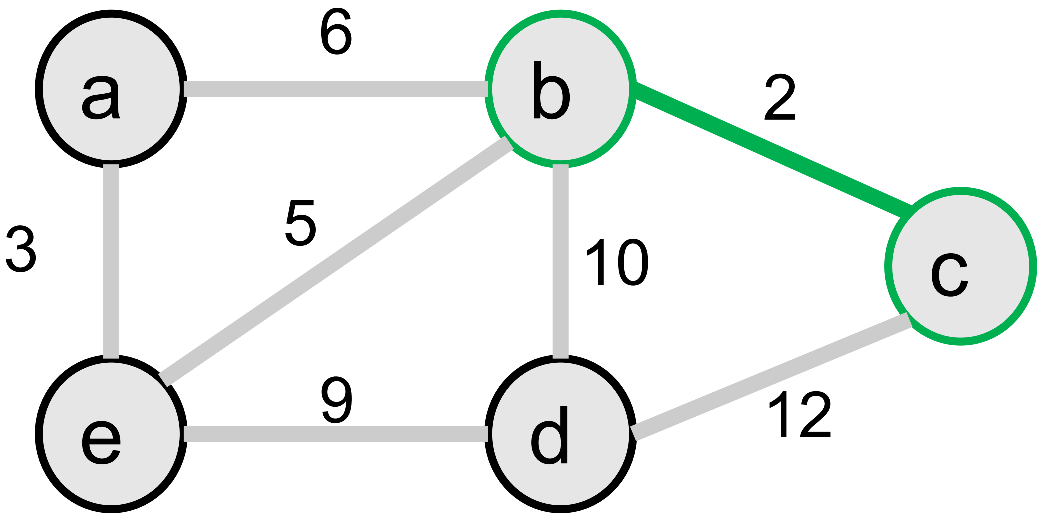
m = |E|

```
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}
```

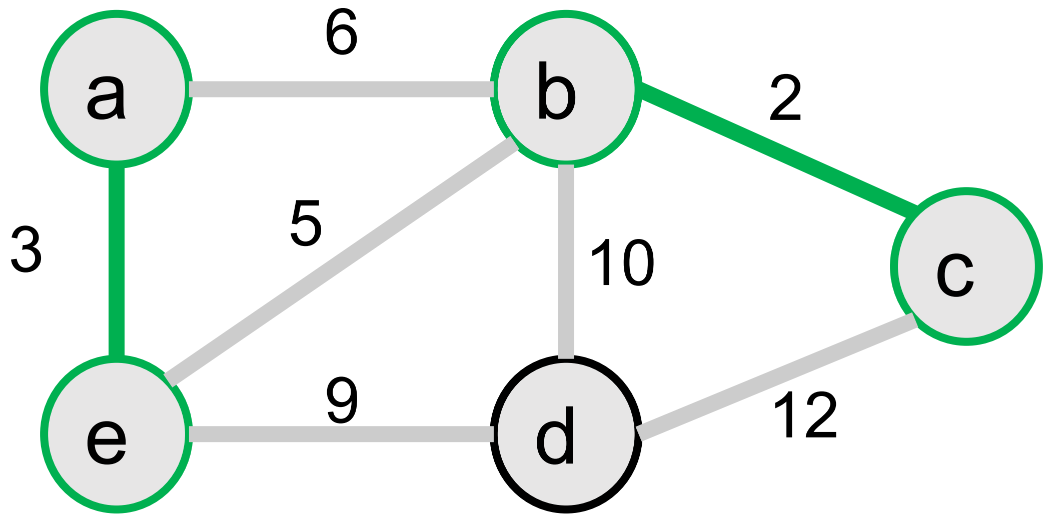
Example



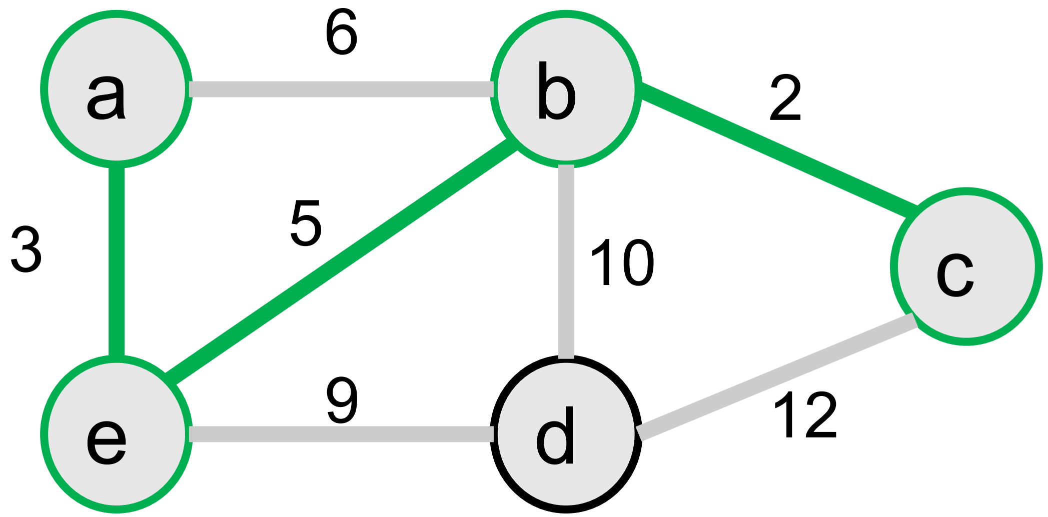
Add (b, c), the lightest edge



Add (a, e), the 2nd lightest

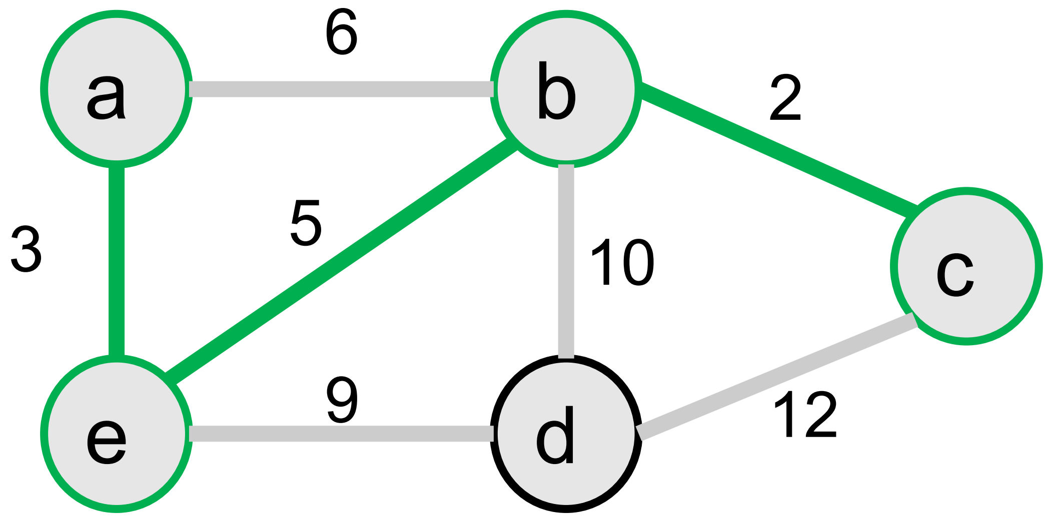


Add (b, e), the 3rd lightest

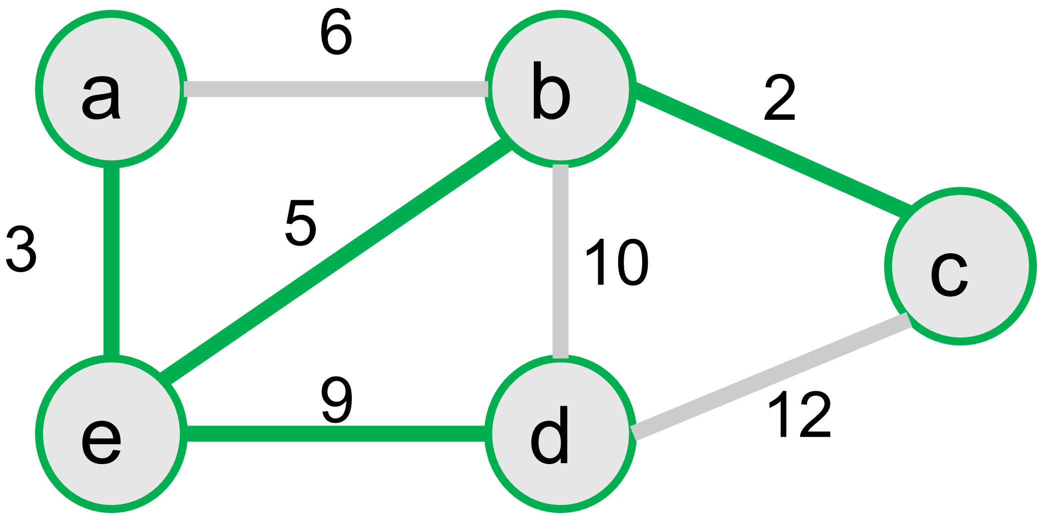


Add (a, b), the 4th lightest ...

No! a, b are in the same component
Add (d, e) instead



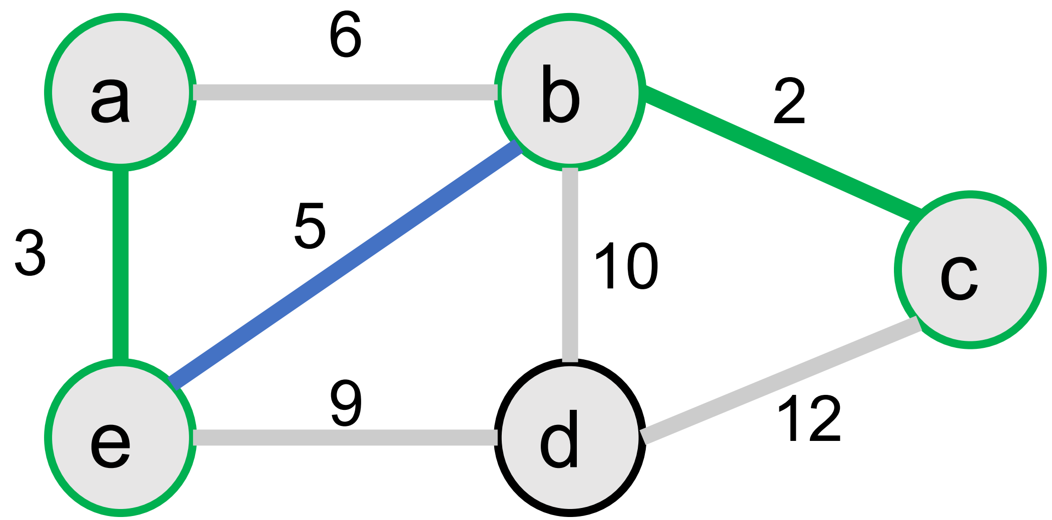
Add (d, e) ... MST grown!



Correctness of Kruskal's

CLRS Theorem 23.1

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 this 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

Disjoint Set