

# Advanced Data Structures and Algorithms

## Minimum Spanning Trees

# THE GREEDY PARADIGM

Commit to choices one-at-a-time,  
never look back,  
and hope for the best.

**Greedy doesn't always work.**  
And when it does, it's not always easy to see & prove why it works.

# A STRATEGY FOR GREEDY PROOFS

Prove that after each choice, you're not ruling out success.  
(i.e. you're not ruling out finding an optimal solution)

- **INDUCTIVE HYPOTHESIS:** After greedy choice  $t$ , you haven't ruled out success
- **BASE CASE:** Success is possible before you make any choices
- **INDUCTIVE STEP:** If you haven't ruled out success after choice  $t$ , then show that you won't rule out success after choice  $t+1$  (there's an optimal solution that's consistent with the choices we've made so far)
- **CONCLUSION:** If you reach the end of the algorithm and haven't ruled out success then you must have succeeded.

# WHAT WE'LL COVER TODAY

- Applications of the greedy algorithm design paradigm to **Minimum Spanning Trees**
  - Prim's algorithm
  - Kruskal's algorithm

# MINIMUM SPANNING TREES

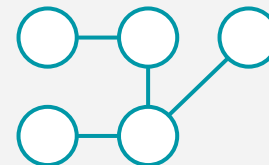
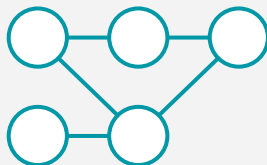
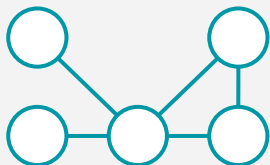
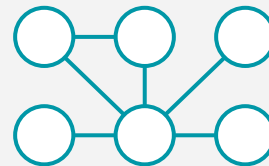
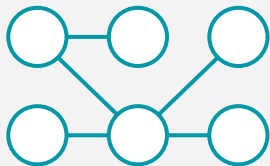
What are minimum spanning trees (MSTs)?

# TREES IN GRAPHS

Let's go over some terminology that we'll be using today.

A tree is an undirected, *acyclic*, connected graph.

Which of these graphs are trees?

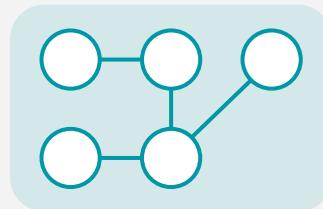
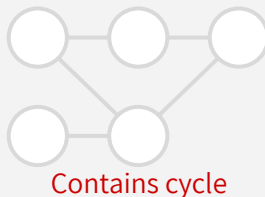
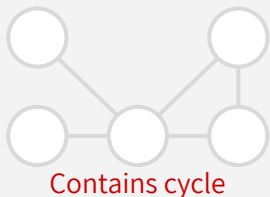
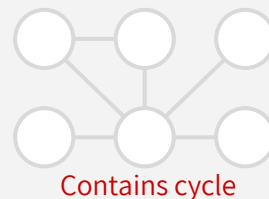
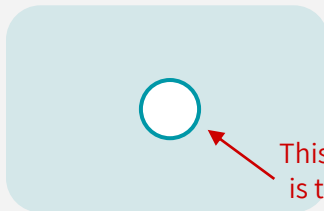
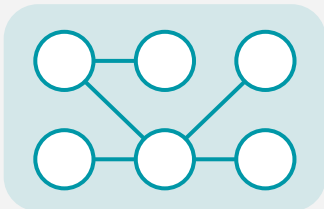


# TREES IN GRAPHS

Let's go over some terminology that we'll be using today.

A tree is an undirected, *acyclic*, connected graph.

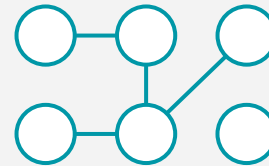
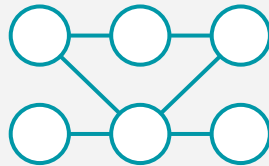
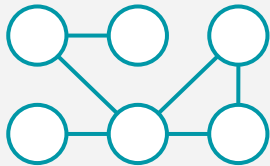
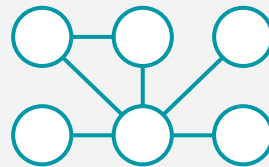
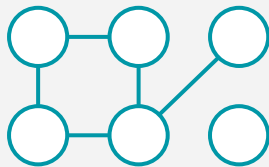
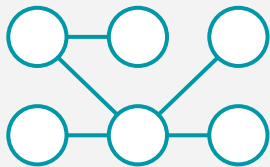
Which of these graphs are trees?



# SPANNING TREES

A spanning tree is a tree that connects all of the vertices in the graph

Which of these graphs are spanning trees?

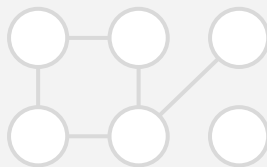
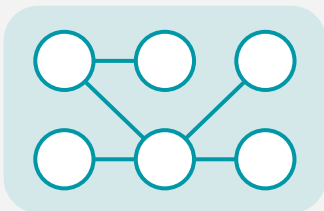




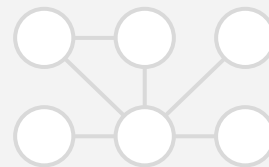
# SPANNING TREES

A spanning tree is a tree that connects all of the vertices

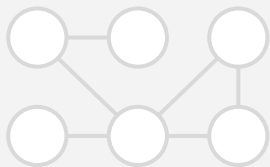
Which of these graphs are spanning trees?



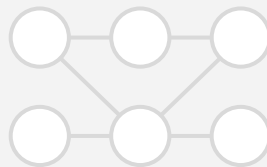
Doesn't connect all vertices



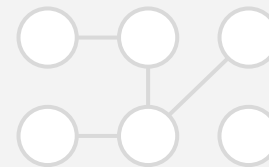
Not a tree



Not a tree



Not a tree



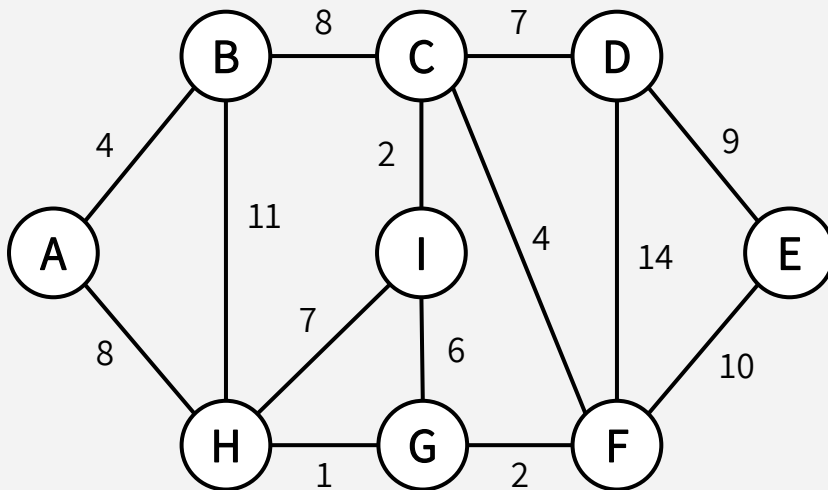
Doesn't connect all vertices

# MINIMUM SPANNING TREES (MSTs)

For the remainder of today, we're going to work with **undirected, weighted, connected graphs**.

The **cost of a spanning tree** is the **sum of the weights on the edges**.

An **MST** of a graph is a spanning tree of the graph with minimum cost.



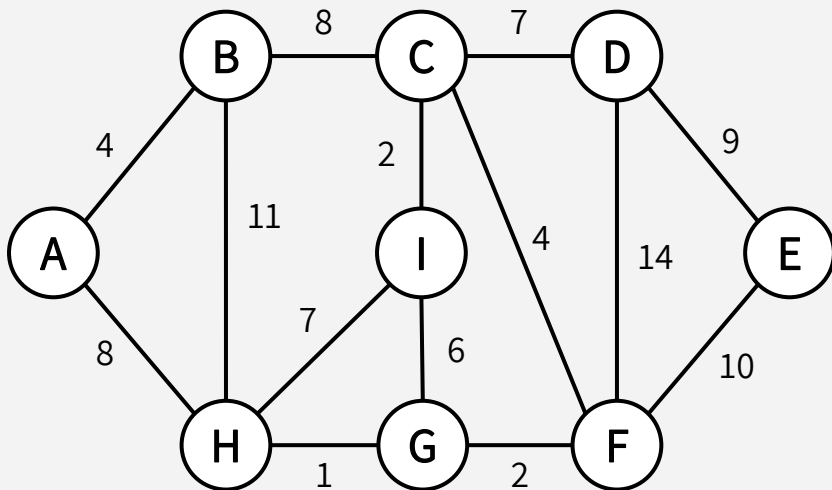
# MINIMUM SPANNING TREES (MSTs)

For the remainder of today, we're going to work with **undirected, weighted, connected** graphs.

The **cost of a spanning tree** is the **sum of the weights on the edges**.

An **MST** of a graph is a spanning tree of the graph with minimum cost.

**Note:** A graph may have multiple spanning trees. It may also have multiple MSTs (if 2 different spanning trees have the same exact cost)



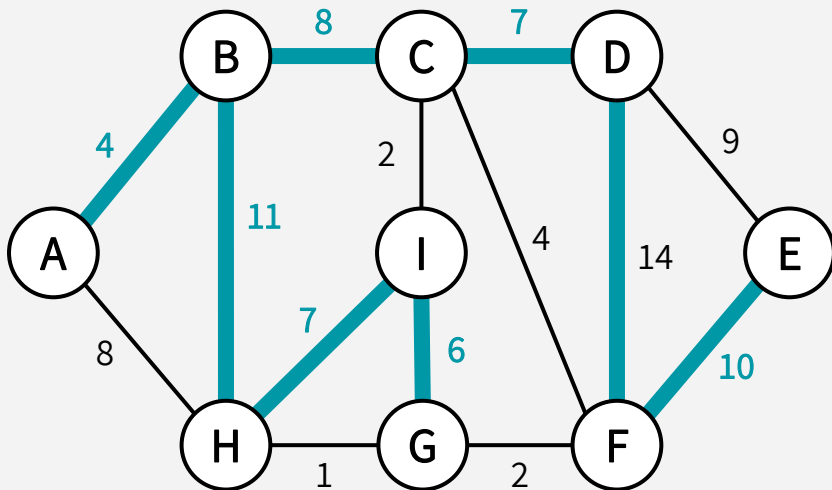
# MINIMUM SPANNING TREES (MSTs)

For the remainder of today, we're going to work with **undirected, weighted, connected graphs**.

The **cost of a spanning tree** is the **sum of the weights on the edges**.

An **MST** of a graph is a spanning tree of the graph with minimum cost.

**Note:** A graph may have multiple spanning trees. It may also have multiple MSTs (if 2 different spanning trees have the same exact cost)



This spanning tree has a cost of **67**.

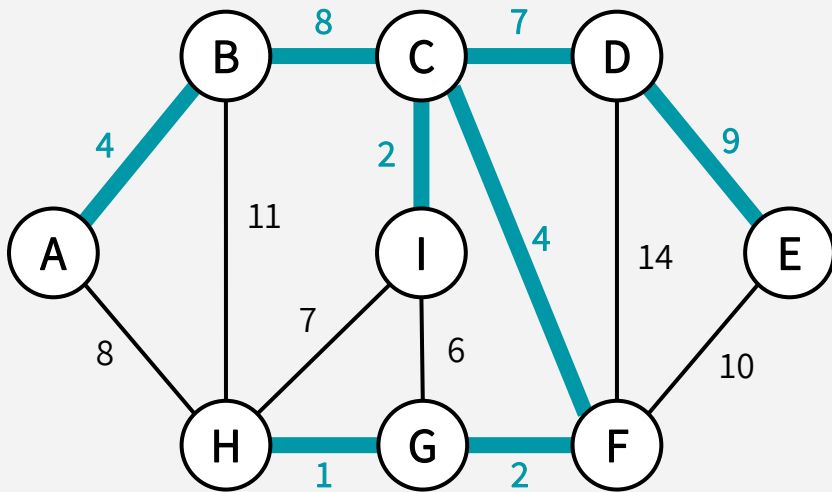
# MINIMUM SPANNING TREES (MSTs)

For the remainder of today, we're going to work with **undirected, weighted, connected graphs**.

The **cost of a spanning tree** is the **sum of the weights on the edges**.

An **MST** of a graph is a spanning tree of the graph with minimum cost.

**Note:** A graph may have multiple spanning trees. It may also have multiple MSTs (if 2 different spanning trees have the same exact cost)



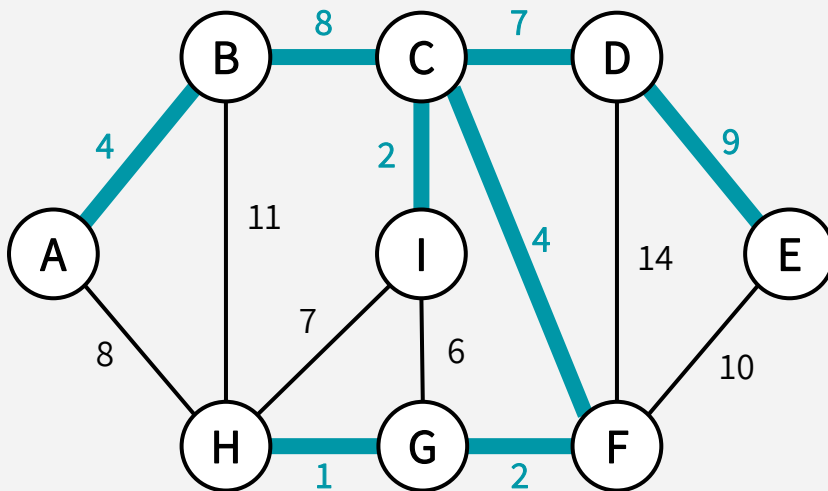
This spanning tree has a cost of 37.

This is an MST of this graph, since there is no other spanning tree with smaller cost.

# MINIMUM SPANNING TREES (MSTs)

## The task for today:

Given an undirected, weighted, and connected graph  $G$ , find the minimum spanning tree (as a subset of the  $G$ 's edges)



We would return this MST. Sometimes, there may be more than one MST as well, so return any MST of  $G$ .

# APPLICATIONS OF MSTs

## Network design

Find the most cost-effective way to connect cities with roads/water/electricity/phone

## Image processing

Image segmentation, which finds connected regions in the image with minimal differences

## Cluster analysis

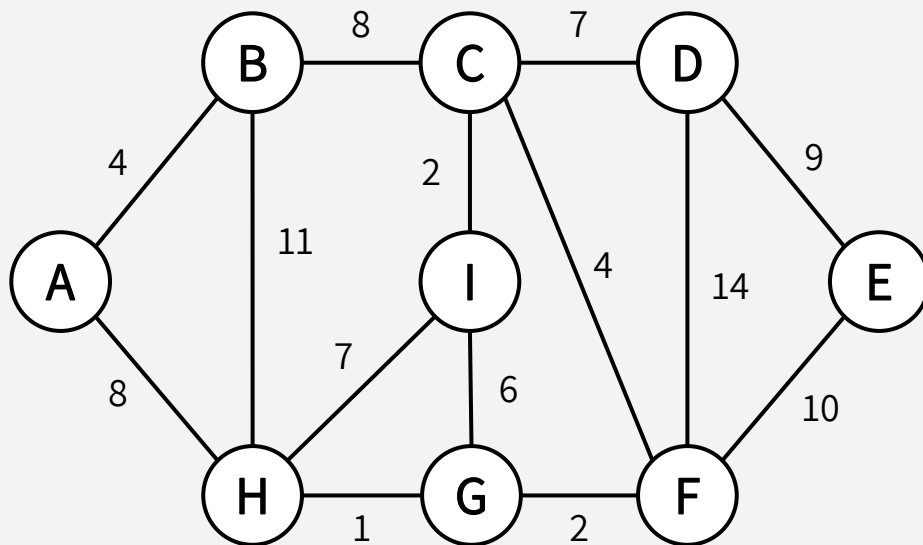
Find clusters in a dataset (one of the algorithms we'll see can be modified slightly to basically do this)

## Useful primitive

Finding an MST is often useful as a subroutine or approximation for more advanced graph algorithms

# MINIMUM SPANNING TREES (MSTs)

Before we move on with the lecture... why don't you give this a try?  
**Brainstorm some greedy algorithms to find an MST!**



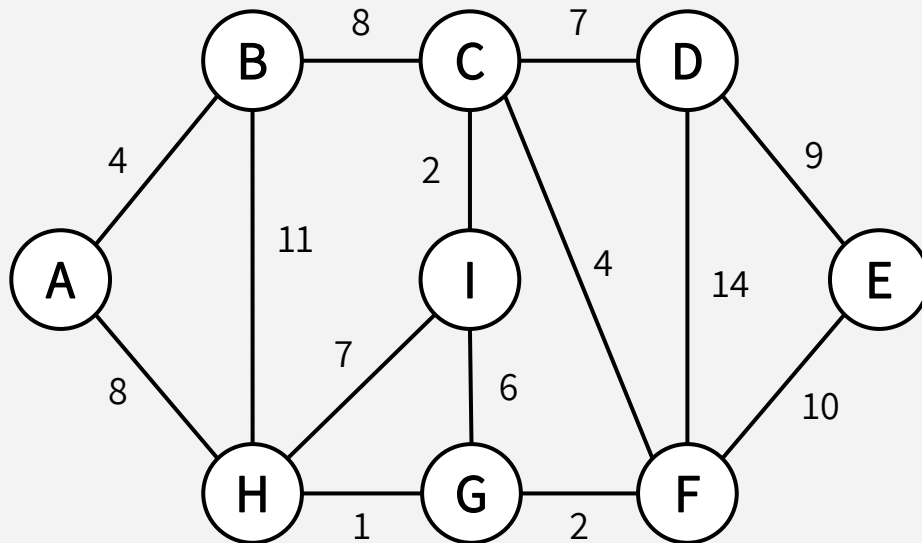


# A BRIEF ASIDE: CUTS & “LIGHT” EDGES

What are cuts in graphs? And what can they tell us about MSTs?

# CUTS IN GRAPHS

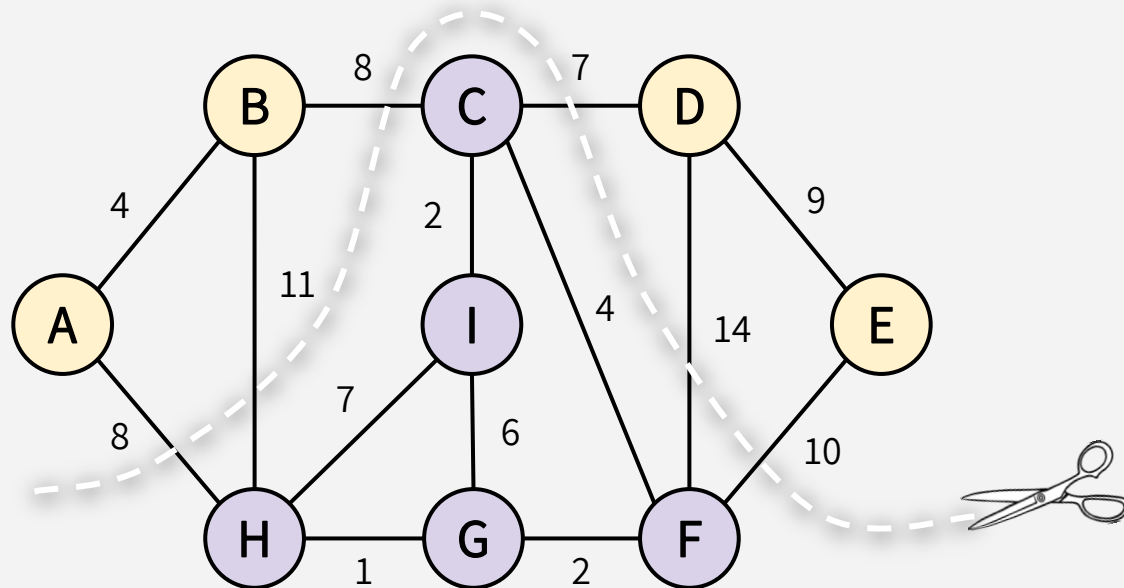
A **cut** is a partition of the vertices into two nonempty parts.



# CUTS IN GRAPHS

A **cut** is a partition of the vertices into two nonempty parts.

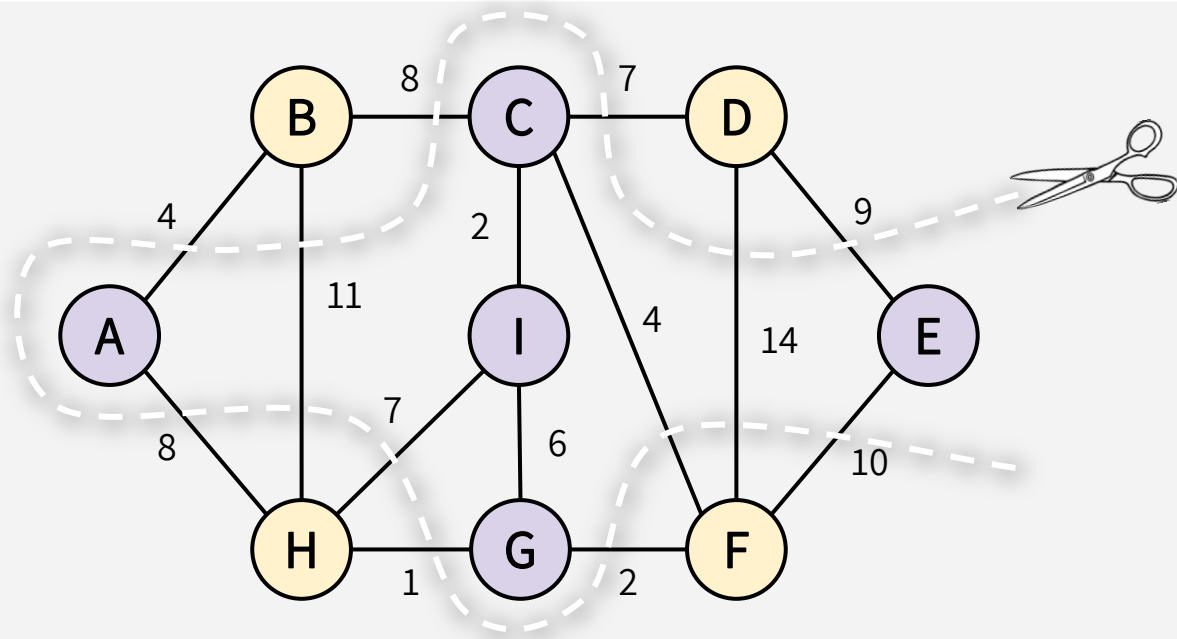
e.g. this is the  
cut “{A,B,D,E}  
and  
{C,I,F,G,H}”



# CUTS IN GRAPHS

A **cut** is a partition of the vertices into two nonempty parts.

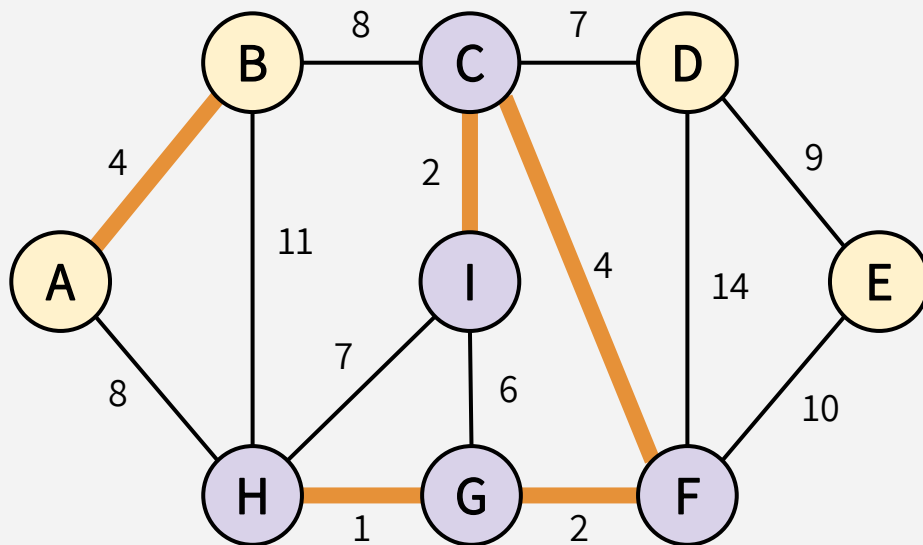
e.g. this is the  
cut “{B,D,H,F}  
and  
{A,C,I,G,E}”



# CUTS IN GRAPHS

A **cut** is a partition of the vertices into two nonempty parts.

We say a **cut respects a set of edges S** if no edges in S cross the cut

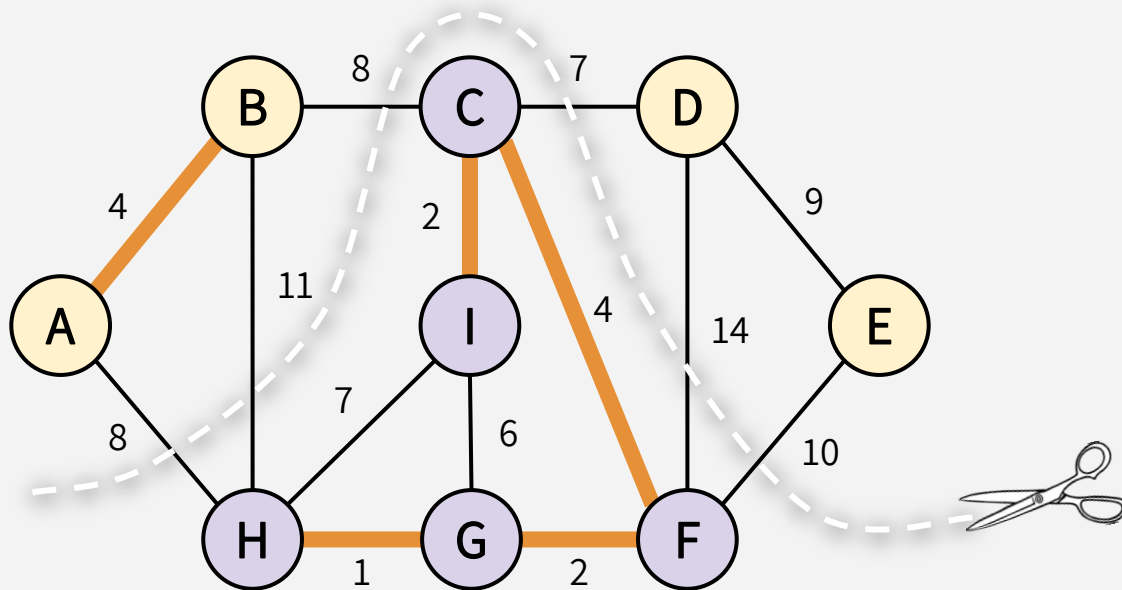


# CUTS IN GRAPHS

A **cut** is a partition of the vertices into two nonempty parts.

We say a **cut respects a set of edges S** if no edges in S cross the cut

This cut  
**respects** this  
orange set of  
edges!

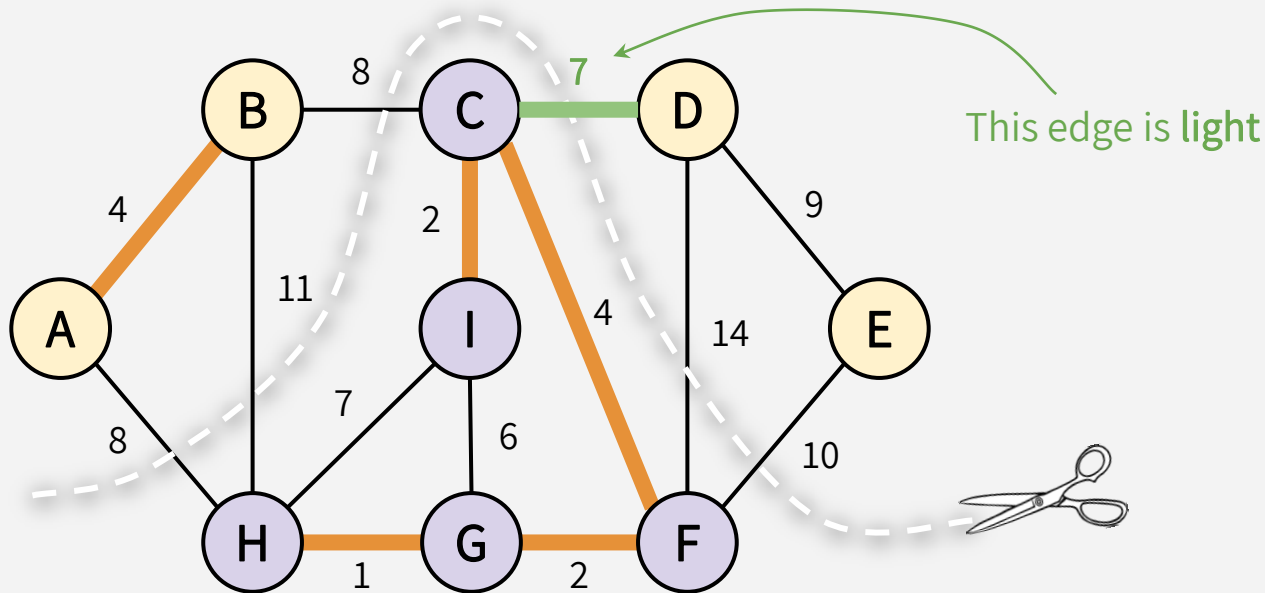


# CUTS IN GRAPHS

A **cut** is a partition of the vertices into two nonempty parts.

An edge is **light** if it has the smallest weight of any edge crossing the cut

This cut  
**respects** this  
orange set of  
edges!

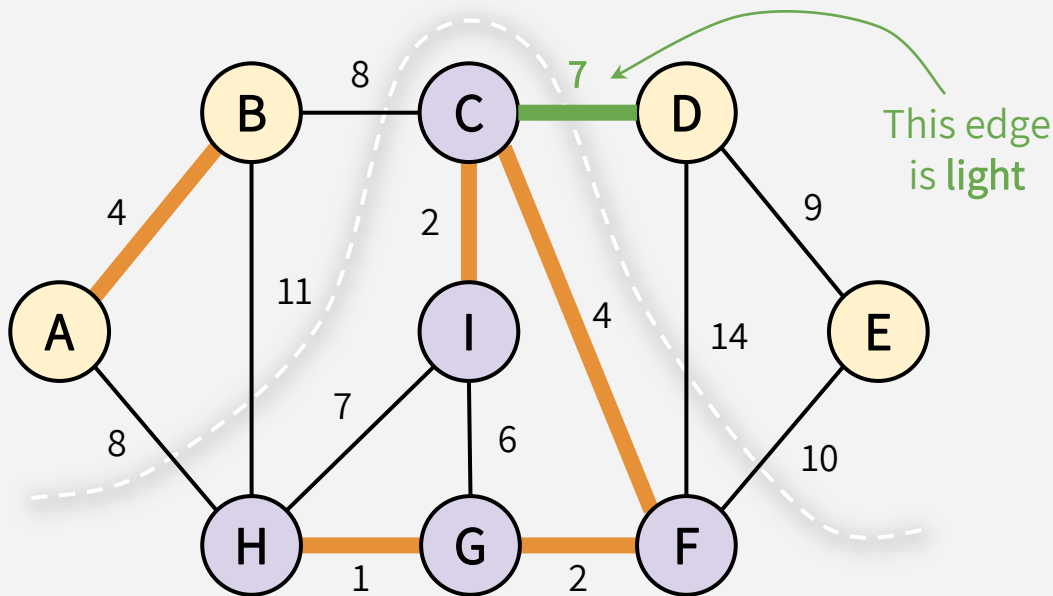


# AN IMPORTANT LEMMA

**LEMMA:** Consider a cut that respects a set of edges  $S$ .

Suppose there exists an MST  $T^*$  containing  $S$ . Let  $(u,v)$  be a light edge crossing this cut.

Then, there exists an MST containing  $S \cup \{(u,v)\}$ .





# AN IMPORTANT LEMMA

**LEMMA:** Consider a cut that respects a set of edges  $S$ .

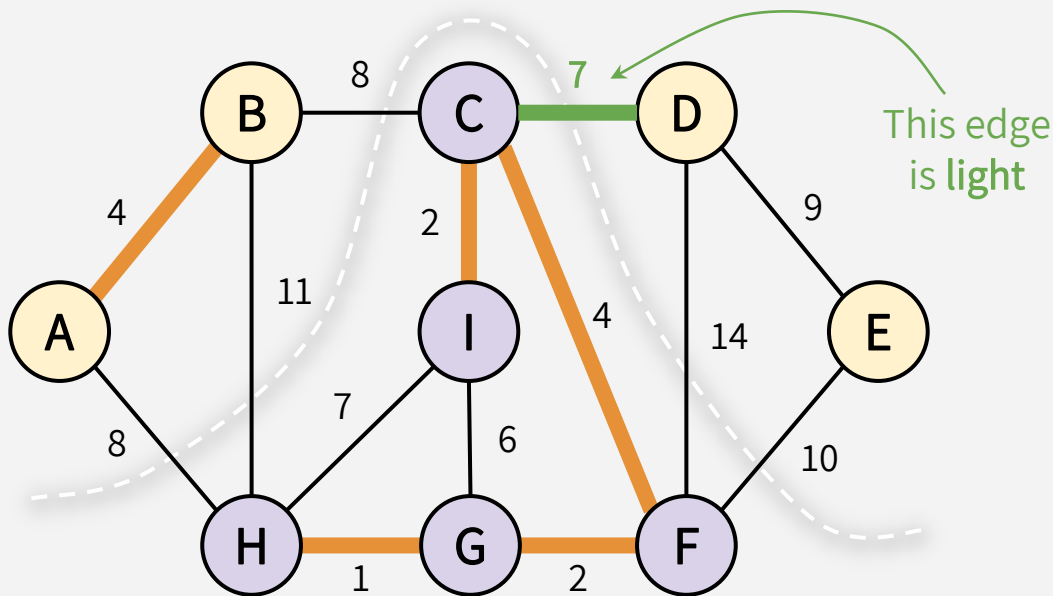
Suppose there exists an MST  $T^*$  containing  $S$ . Let  $(u,v)$  be a light edge crossing this cut.

Then, there exists an MST containing  $S \cup \{(u,v)\}$ .

Before we prove this, why is  
this lemma important?

This is exactly the kind of statement  
we want for a greedy algorithm: *If we  
haven't ruled out the possibility of  
success so far, then adding a light  
edge still won't rule out success!*

We'll see how this can translate to an  
algorithm later... let's prove this first!



# PROOF OF LEMMA

**LEMMA:** Consider a cut that respects a set of edges  $S$ .

Suppose there exists an MST  $T^*$  containing  $S$ . Let  $(u,v)$  be a light edge crossing this cut.

Then, there exists an MST containing  $S \cup \{(u,v)\}$ .

Suppose  $(u,v)$  is not in  $T^*$ .

If it is, then trivially,  $T^*$  is an MST containing  $S \cup \{(u,v)\}$ .

Adding  $(u,v)$  to  $T^*$  will make a cycle.

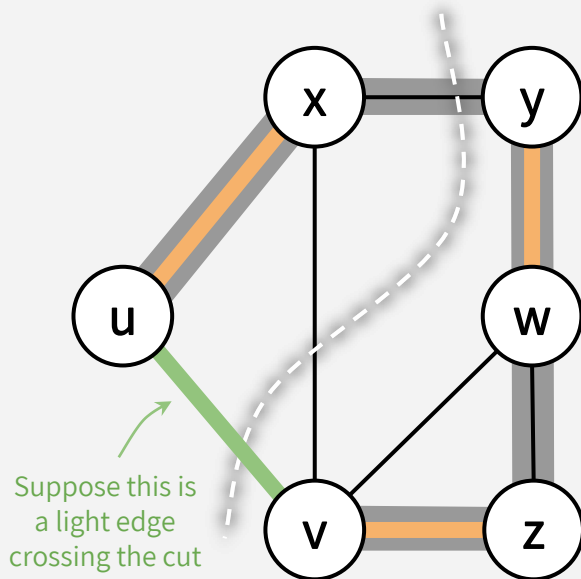
Since  $T^*$  is an MST, we know that there must be an edge in  $T^*$  crossing the cut (in order to connect nodes on opposite sides of the cut).

Call this edge  $(x,y)$ .

If we replace  $(x,y)$  with  $(u,v)$  in  $T^*$ , we end up with an MST  $T$ .

Why is  $T$  still an MST? Well, since  $T^*$  was a tree, and we also delete  $(x,y)$ , then  $T$  must also be a tree (no cycles). Since  $(u,v)$  is light, then  $T$  has at most the cost of  $T^*$ , so  $T$  is also optimal.

Thus, there exists an MST (T) containing  $S \cup \{(u,v)\}$



# AN IMPORTANT LEMMA

**LEMMA:** Consider a cut that respects a set of edges  $S$ .

Suppose there exists an MST  $T^*$  containing  $S$ . Let  $(u,v)$  be a light edge crossing this cut.

Then, there exists an MST containing  $S \cup \{(u,v)\}$ .

**We'll see two famous MST algorithms which each have their own way of greedily claiming the next light edge.**

**We'll keep this lemma in mind when working out the proofs of correctness for each of the algorithms!**

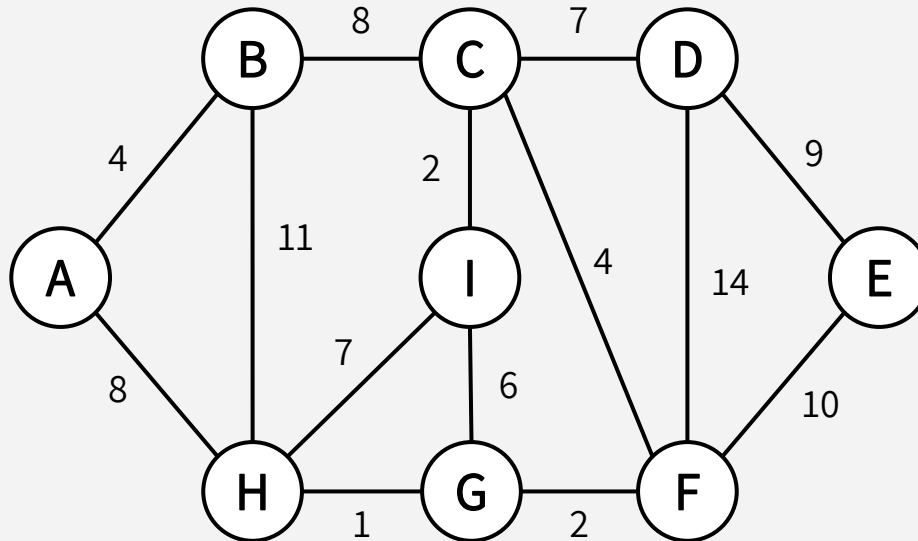
# PRIM'S ALGORITHM

Greedyly add the closest vertex!

# PRIM'S ALGORITHM: THE IDEA

## Greedy choice:

Grow a single tree, & greedily add the shortest edge that could grow our tree

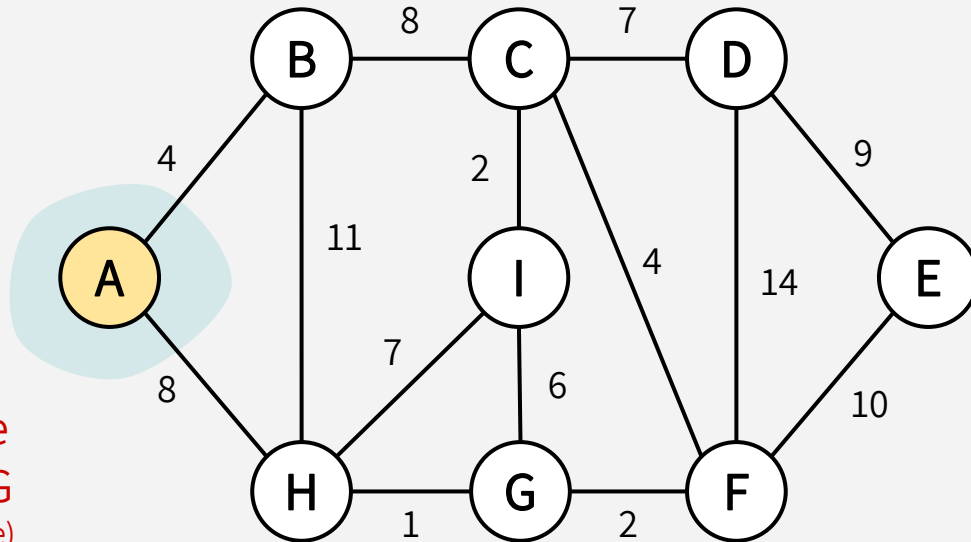


# PRIM'S ALGORITHM: THE IDEA

## Greedy choice:

Grow a single tree, & greedily add the shortest edge that could grow our tree

First, we can  
initialize our tree  
to contain a single  
arbitrary node in  $G$   
(doesn't matter which node)

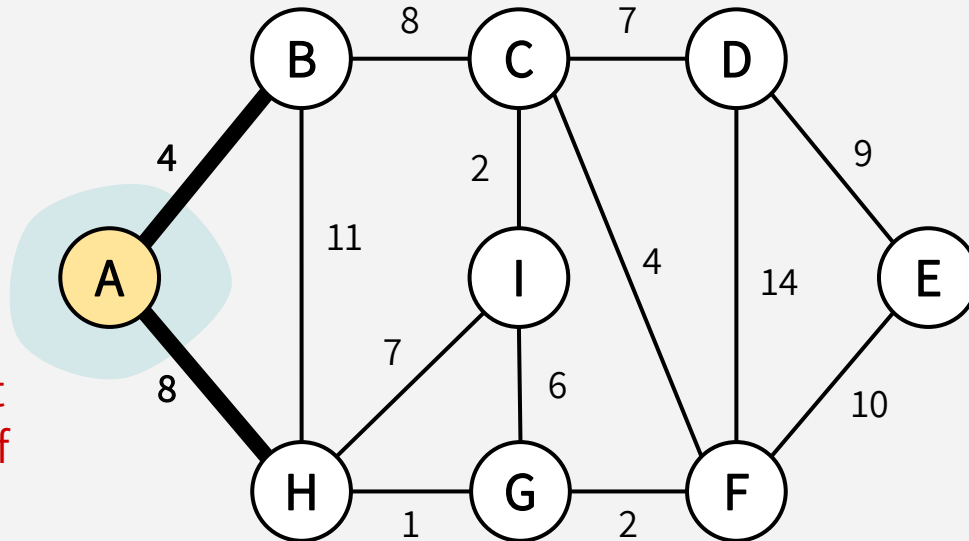


# PRIM'S ALGORITHM: THE IDEA

## Greedy choice:

Grow a single tree, & greedily add the shortest edge that could grow our tree

Consider the edges coming out of the “frontier” of our growing tree.

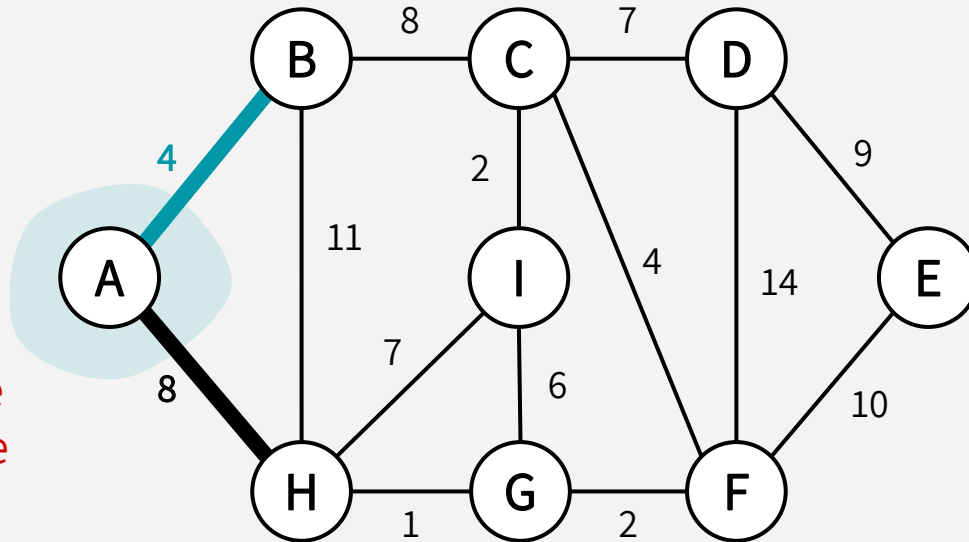


# PRIM'S ALGORITHM: THE IDEA

## Greedy choice:

Grow a single tree, & greedily add the shortest edge that could grow our tree

Claim the edge coming out of the “frontier” with the smallest weight

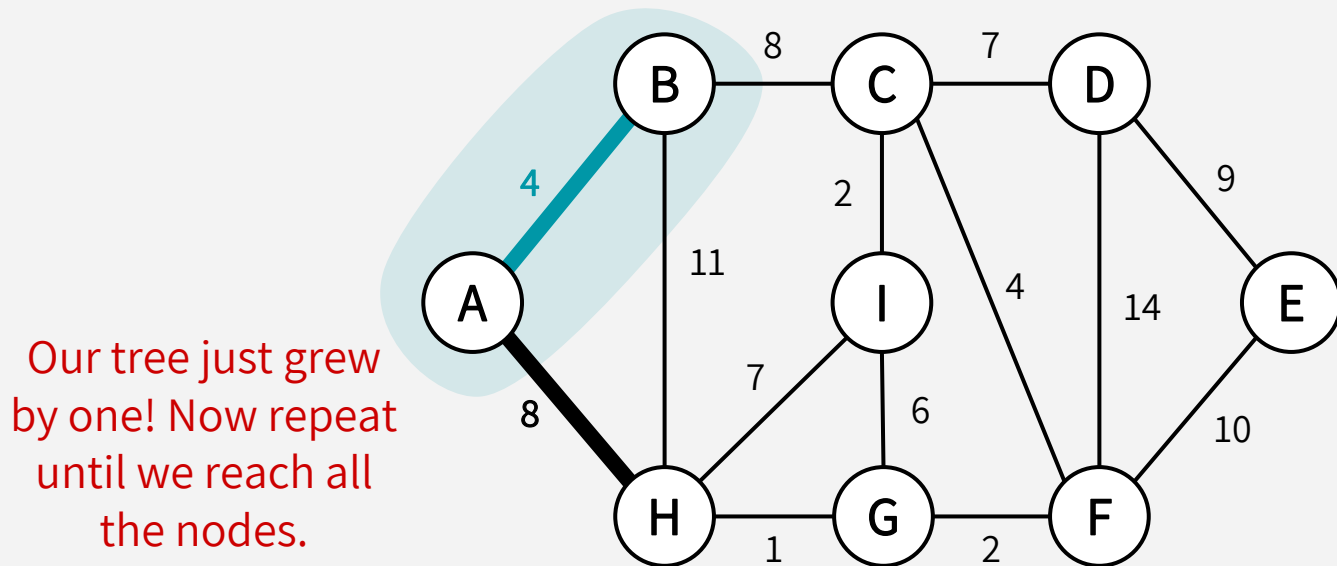




# PRIM'S ALGORITHM: THE IDEA

## Greedy choice:

Grow a single tree, & greedily add the shortest edge that could grow our tree

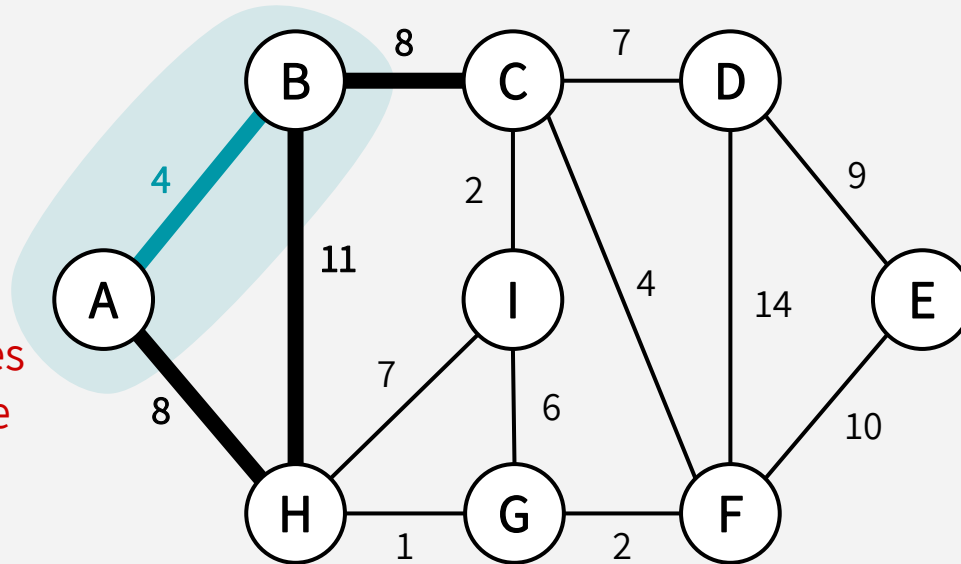


# PRIM'S ALGORITHM: THE IDEA

## Greedy choice:

Grow a single tree, & greedily add the shortest edge that could grow our tree

Consider the edges coming out of the “frontier” of our growing tree.

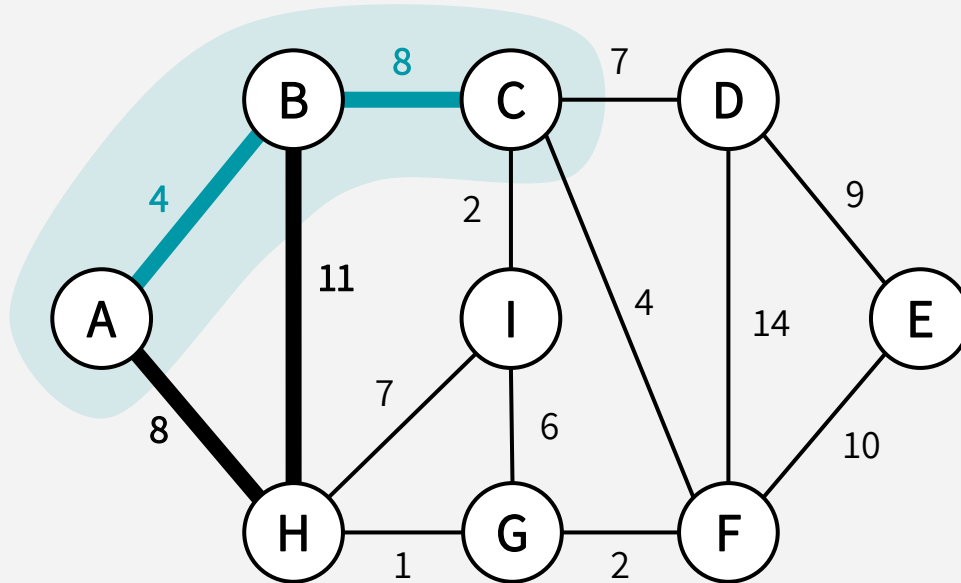


# PRIM'S ALGORITHM: THE IDEA

## Greedy choice:

Grow a single tree, & greedily add the shortest edge that could grow our tree

Claim the edge coming out of the “frontier” with the smallest weight (if there’s a tie, choose any)

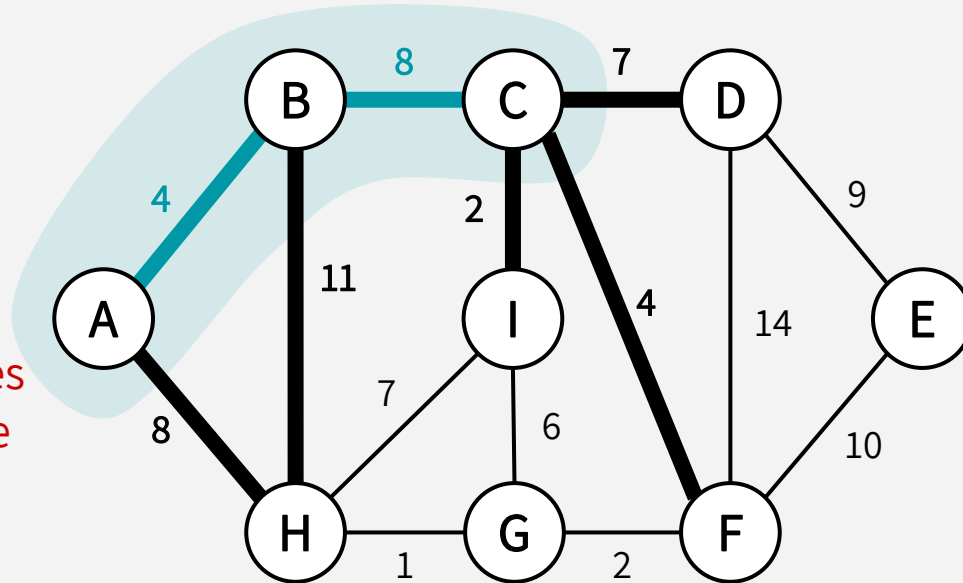


# PRIM'S ALGORITHM: THE IDEA

## Greedy choice:

Grow a single tree, & greedily add the shortest edge that could grow our tree

Consider the edges coming out of the “frontier” of our growing tree.

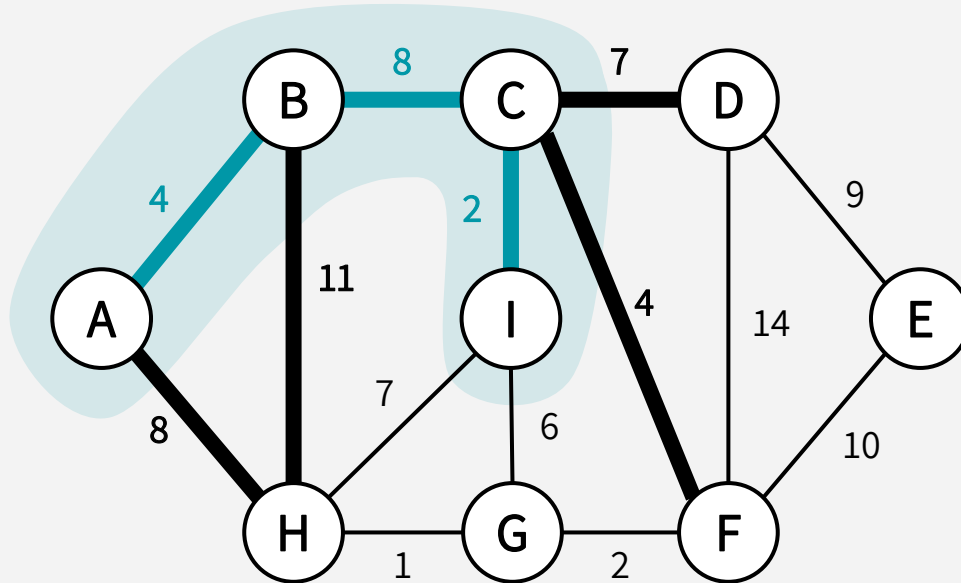


# PRIM'S ALGORITHM: THE IDEA

## Greedy choice:

Grow a single tree, & greedily add the shortest edge that could grow our tree

Claim the edge coming out of the “frontier” with the smallest weight

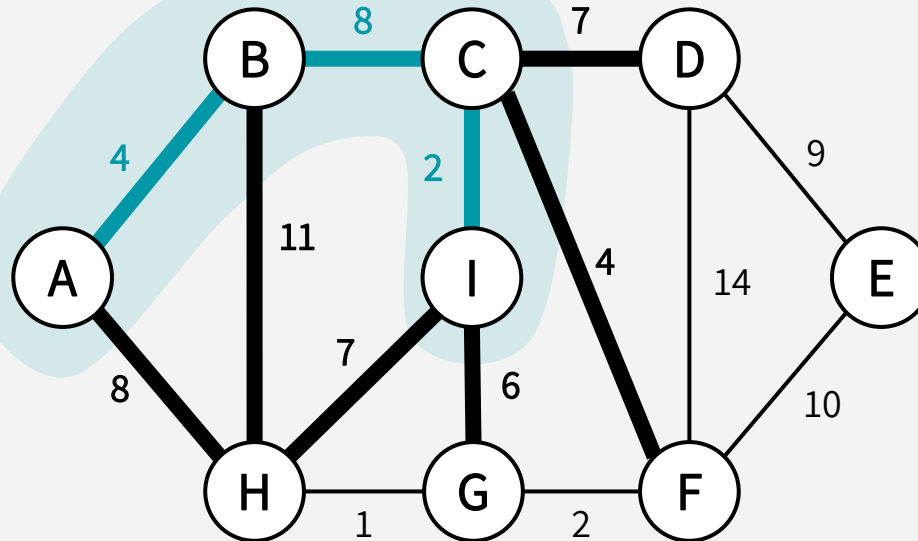


# PRIM'S ALGORITHM: THE IDEA

## Greedy choice:

Grow a single tree, & greedily add the shortest edge that could grow our tree

Consider the edges coming out of the “frontier” of our growing tree.

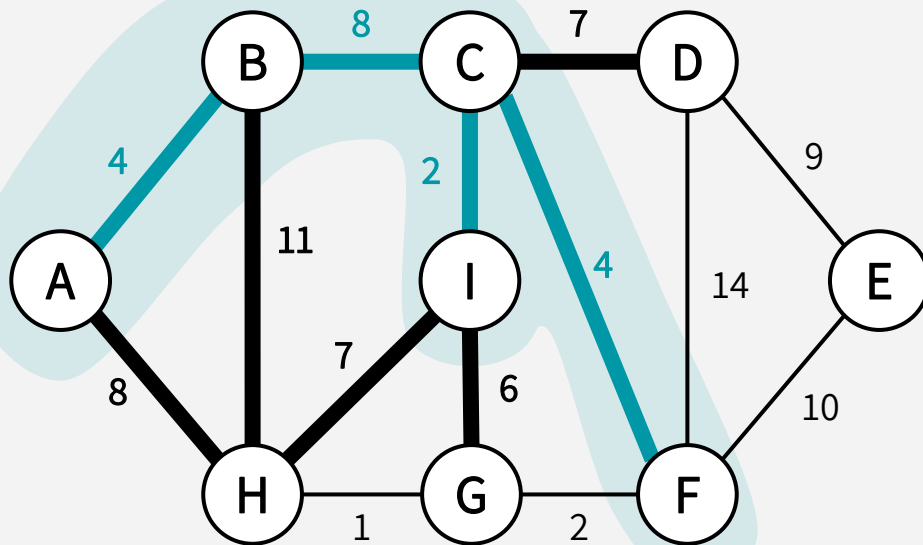


# PRIM'S ALGORITHM: THE IDEA

## Greedy choice:

Grow a single tree, & greedily add the shortest edge that could grow our tree

Claim the edge coming out of the “frontier” with the smallest weight

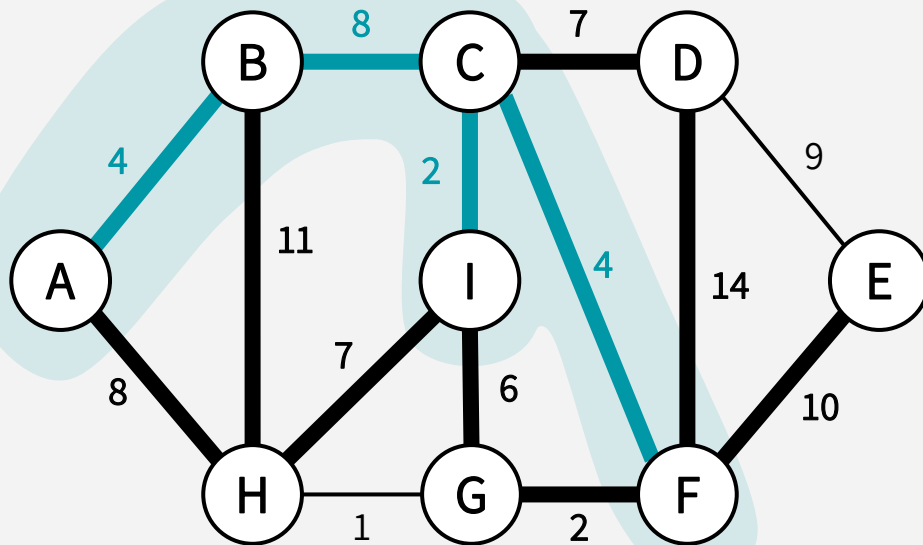


# PRIM'S ALGORITHM: THE IDEA

## Greedy choice:

Grow a single tree, & greedily add the shortest edge that could grow our tree

Consider the edges coming out of the “frontier” of our growing tree.



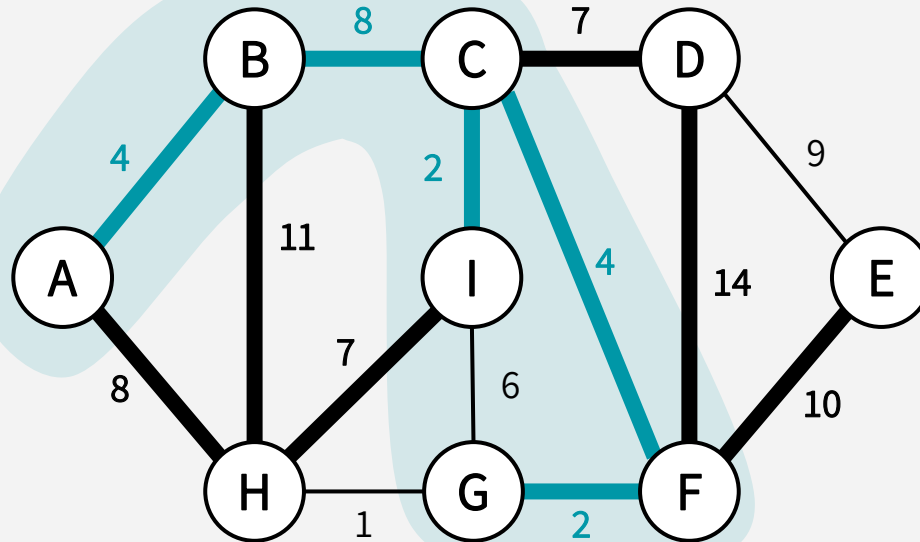


# PRIM'S ALGORITHM: THE IDEA

## Greedy choice:

Grow a single tree, & greedily add the shortest edge that could grow our tree

Claim the edge coming out of the “frontier” with the smallest weight

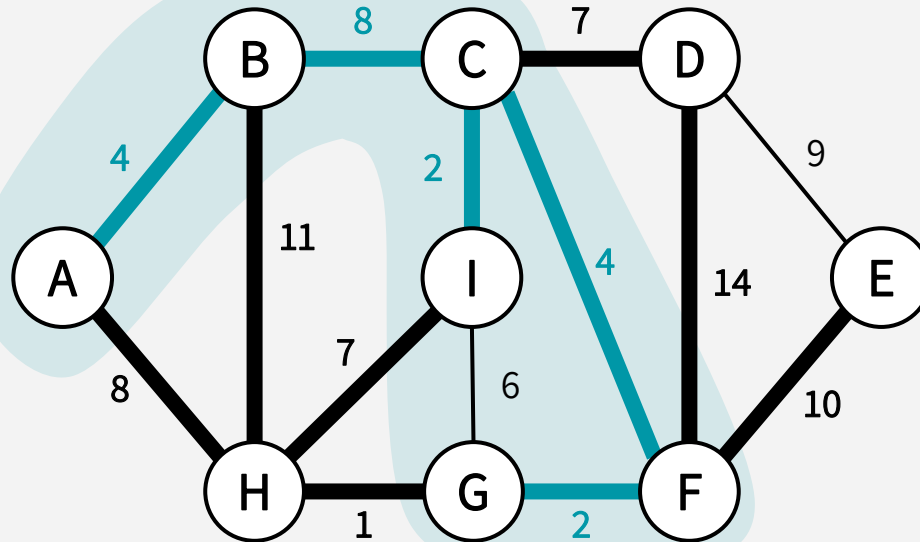


# PRIM'S ALGORITHM: THE IDEA

## Greedy choice:

Grow a single tree, & greedily add the shortest edge that could grow our tree

Consider the edges coming out of the “frontier” of our growing tree.

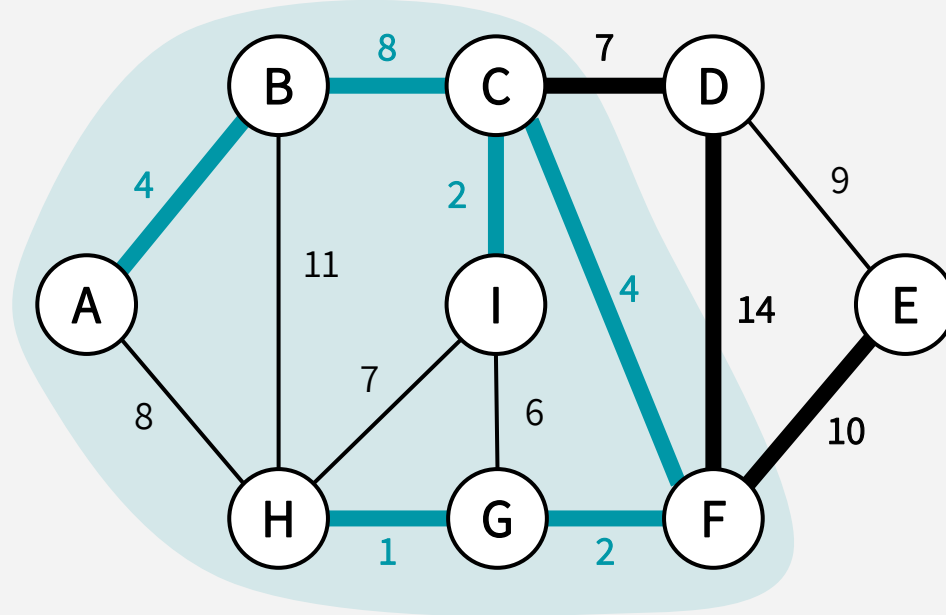


# PRIM'S ALGORITHM: THE IDEA

## Greedy choice:

Grow a single tree, & greedily add the shortest edge that could grow our tree

Claim the edge coming out of the “frontier” with the smallest weight

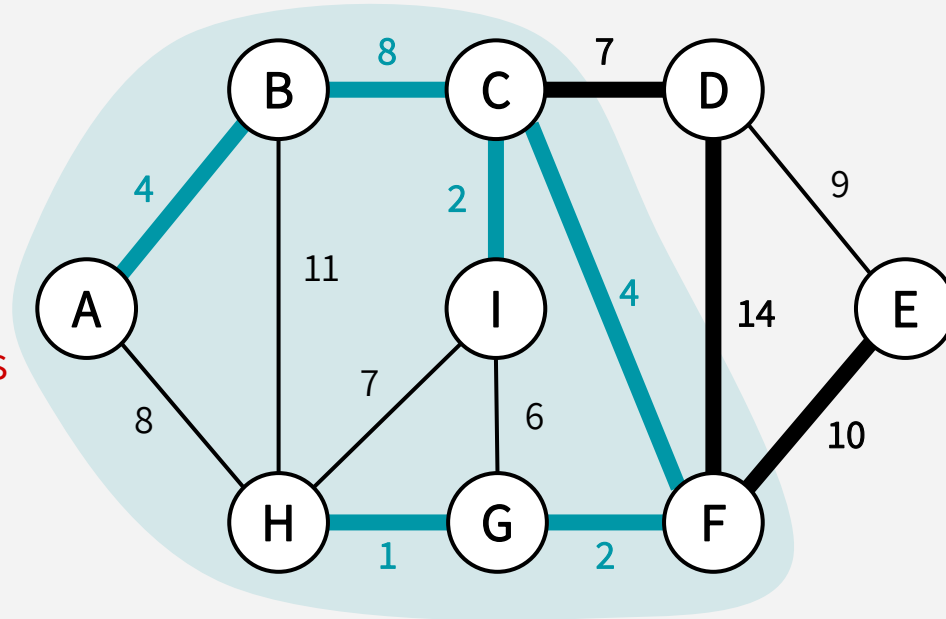


# PRIM'S ALGORITHM: THE IDEA

## Greedy choice:

Grow a single tree, & greedily add the shortest edge that could grow our tree

Consider the edges coming out of the “frontier” of our growing tree.

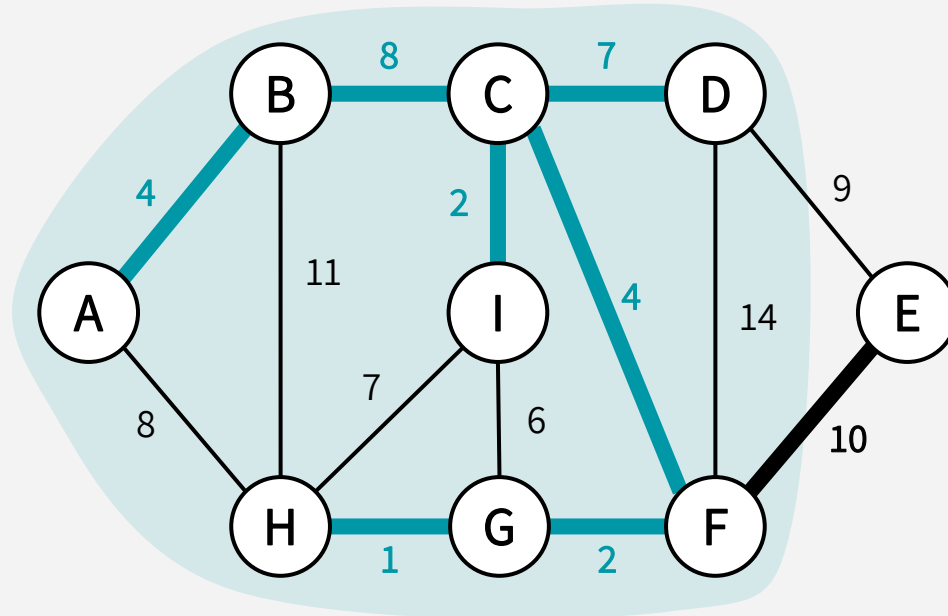


# PRIM'S ALGORITHM: THE IDEA

## Greedy choice:

Grow a single tree, & greedily add the shortest edge that could grow our tree

Claim the edge coming out of the “frontier” with the smallest weight

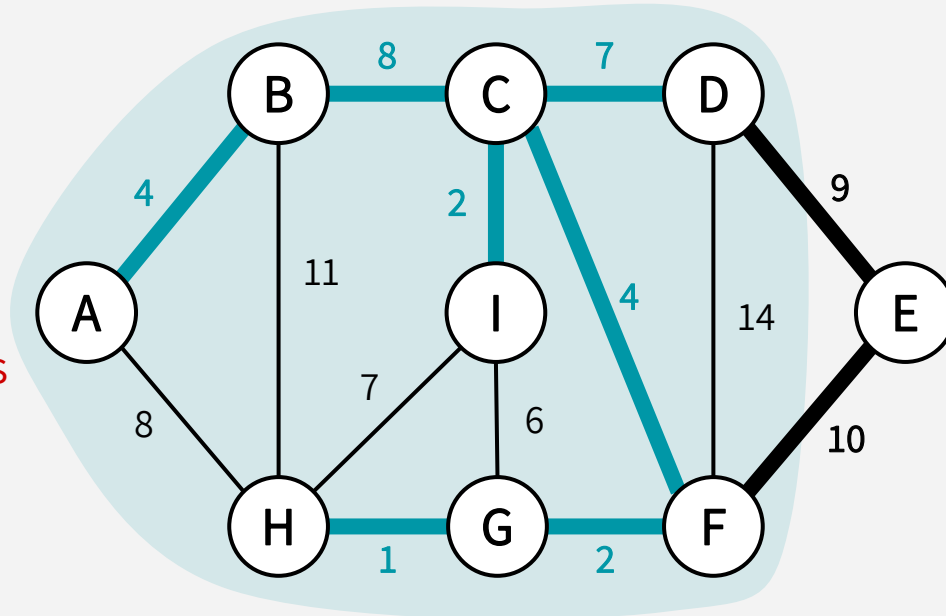


# PRIM'S ALGORITHM: THE IDEA

## Greedy choice:

Grow a single tree, & greedily add the shortest edge that could grow our tree

Consider the edges coming out of the “frontier” of our growing tree.

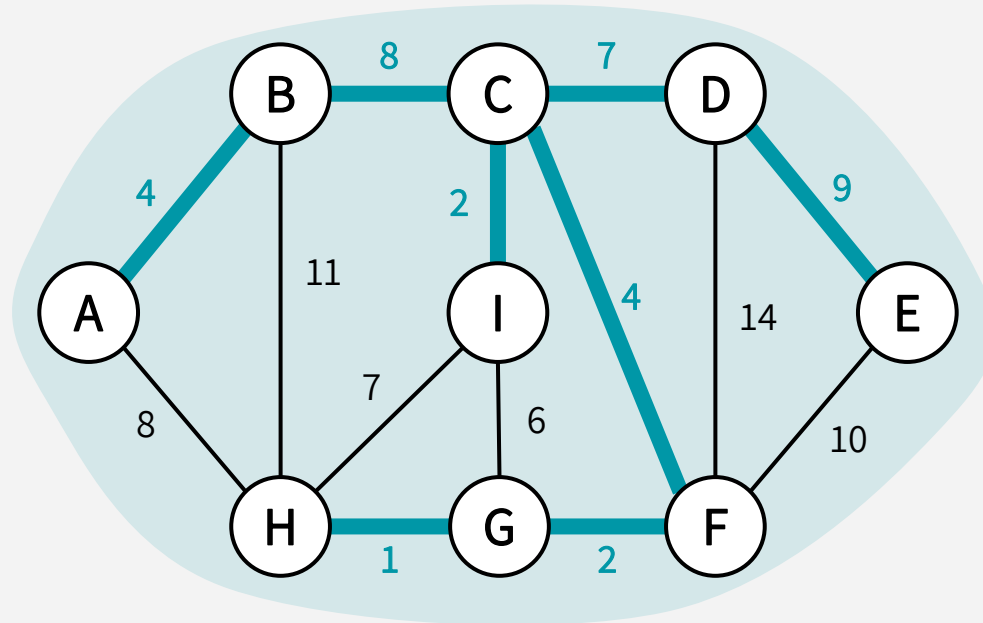


# PRIM'S ALGORITHM: THE IDEA

## Greedy choice:

Grow a single tree, & greedily add the shortest edge that could grow our tree

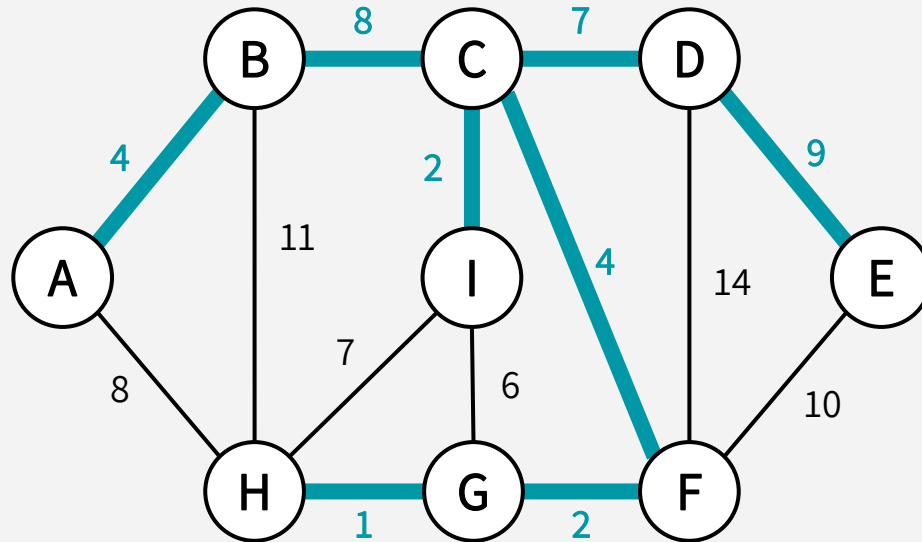
Claim the edge coming out of the “frontier” with the smallest weight



# PRIM'S ALGORITHM: THE IDEA

## Greedy choice:

Grow a single tree, & greedily add the shortest edge that could grow our tree



And we're done!  
**This is our MST.**  
(with weight 37)



# PRIM'S ALGORITHM: SLOW VERSION

```
NAIVE_PRIM(G = (V,E), s):  
    MST = {}  
    visited = {s}  
    while len(visited) < n:  
        find the lightest edge (x,v) in E s.t.  
            • x in visited  
            • v not in visited  
        MST.add((x,v))  
        visited.add(v)  
    return MST
```

If we manually find the lightest edge each iteration, it could be  $O(m)$  time per iteration..

**(Naive) Runtime:  $O(nm)$**

(We'll speed this up by using smart data structures...)

# PRIM'S ALGORITHM: SLOW VERSION

```
NAIVE_PRIM(G = (V,E), s):
```

```
    MST = {}
```

How should we actually implement this?

Each vertex that's not yet reached by the growing tree keeps track of:

- 1) the **distance** from itself to the growing spanning tree using *one edge*
- 2) **how to get to there** (the closest neighbor that's reached by the tree already)

```
    return MST
```

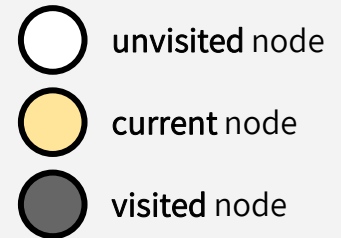
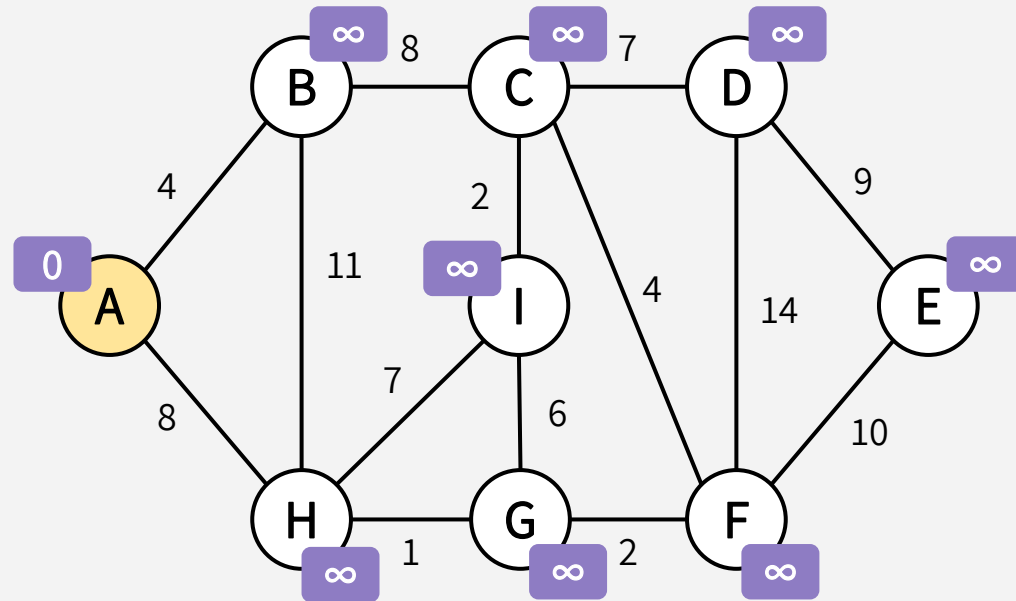
**(Naive) Runtime:  $O(nm)$**

(We'll speed this up by using smart data structures...)

# HOW DO WE IMPLEMENT THIS?

Each vertex that's not yet reached by the growing tree keeps track of:

- 1) the **distance** from itself to the growing spanning tree using *one edge*
- 2) **how to get to there** (the closest neighbor that's reached by the tree already)



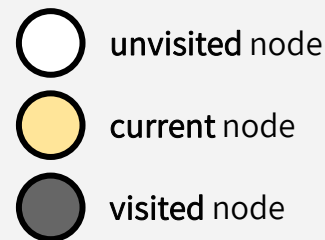
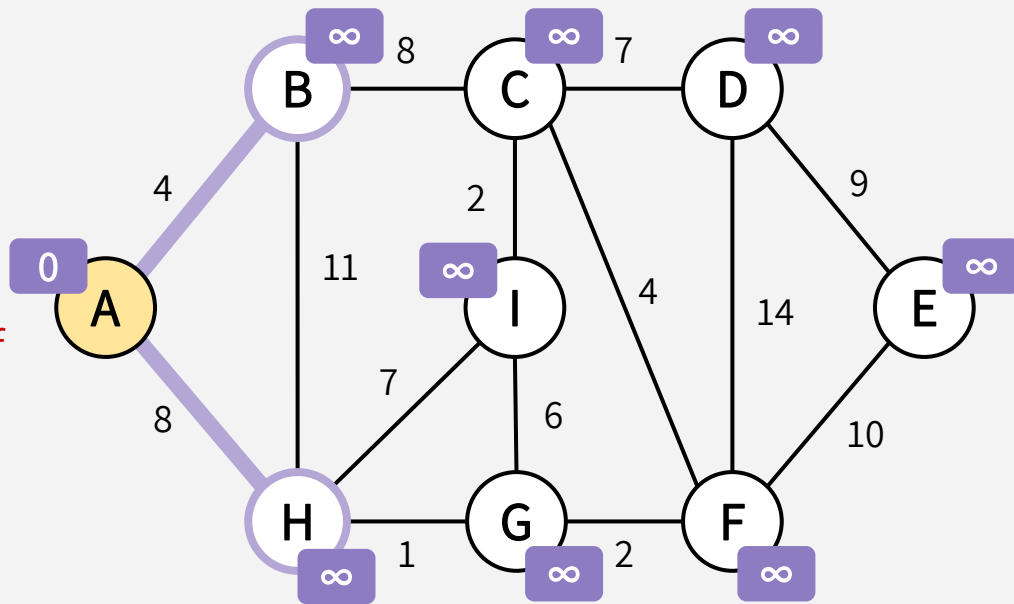
A is part of the  
growing tree first

# HOW DO WE IMPLEMENT THIS?

Each vertex that's not yet reached by the growing tree keeps track of:

- 1) the **distance** from itself to the growing spanning tree using *one edge*
- 2) **how to get to there** (the closest neighbor that's reached by the tree already)

Now that A got added, see if any of its neighbors are closer to the tree because of it!



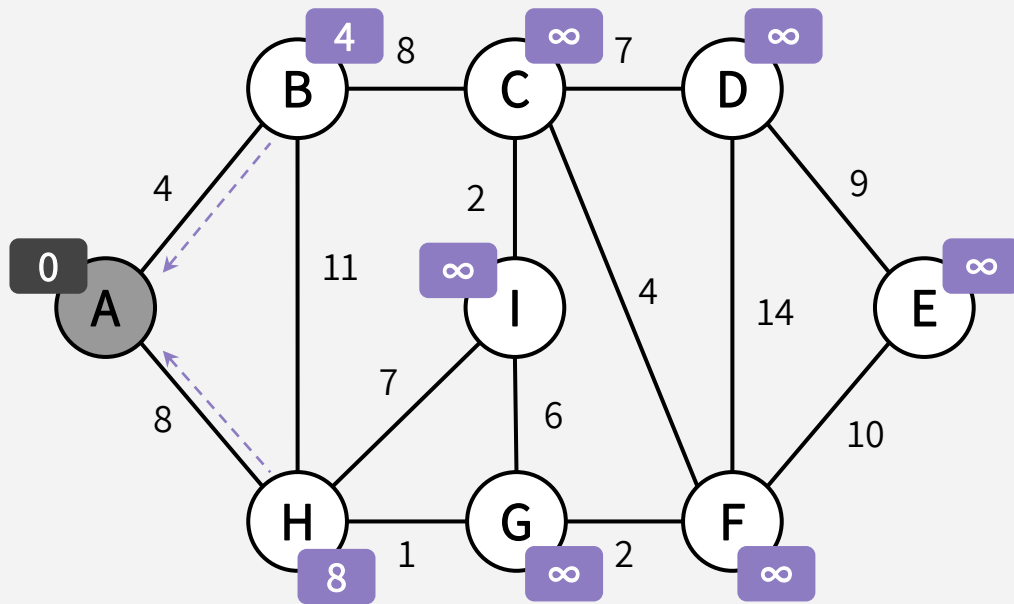
# HOW DO WE IMPLEMENT THIS?

Each vertex that's not yet reached by the growing tree keeps track of:

- 1) the **distance** from itself to the growing spanning tree using *one edge*
- 2) **how to get to there** (the closest neighbor that's reached by the tree already)

Update their estimates, and now A is officially done.

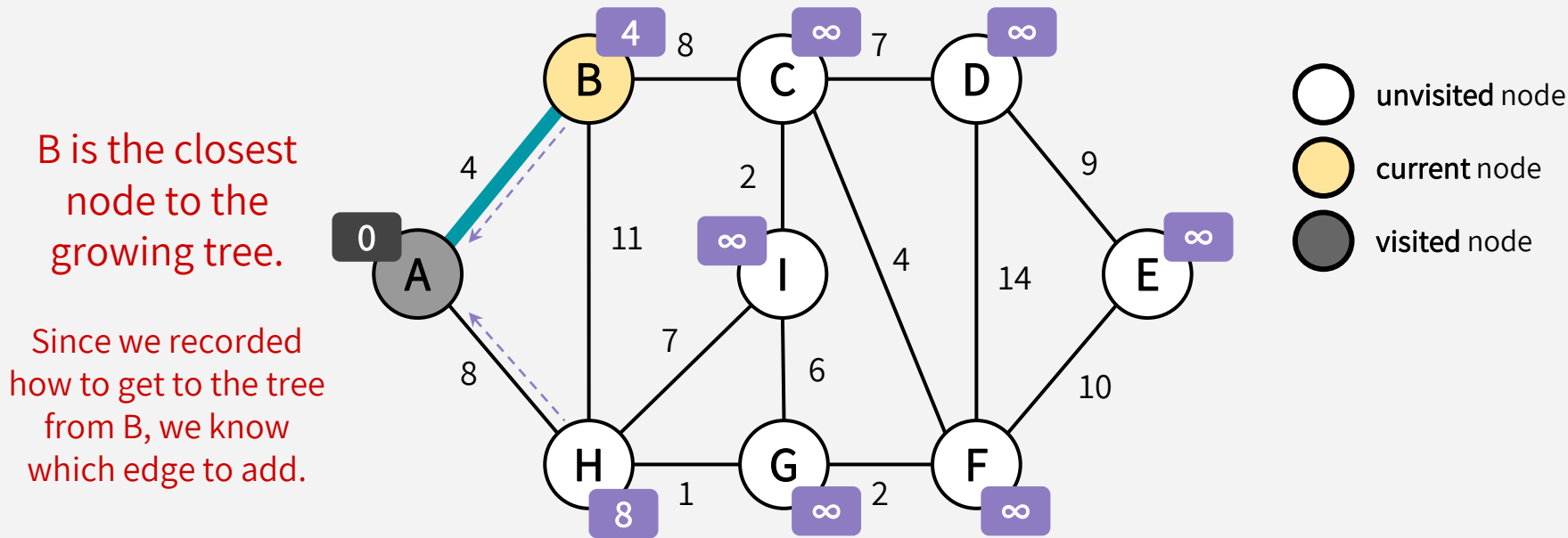
Time to choose the lightest edge on the frontier (i.e. the edge whose endpoint has the lowest distance stored)



# HOW DO WE IMPLEMENT THIS?

Each vertex that's not yet reached by the growing tree keeps track of:

- 1) the **distance** from itself to the growing spanning tree using *one edge*
- 2) **how to get to there** (the closest neighbor that's reached by the tree already)

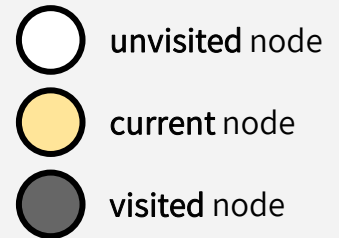
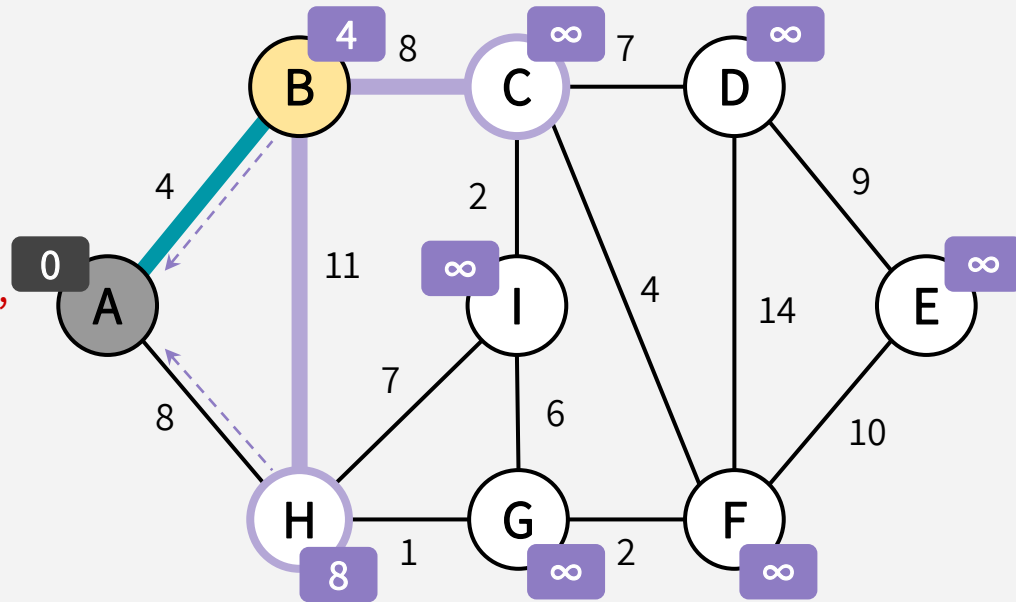


# HOW DO WE IMPLEMENT THIS?

Each vertex that's not yet reached by the growing tree keeps track of:

- 1) the **distance** from itself to the growing spanning tree using *one edge*
- 2) **how to get to there** (the closest neighbor that's reached by the tree already)

Now that B is reached by the tree, see if any of its neighbors are closer to the tree because of it!



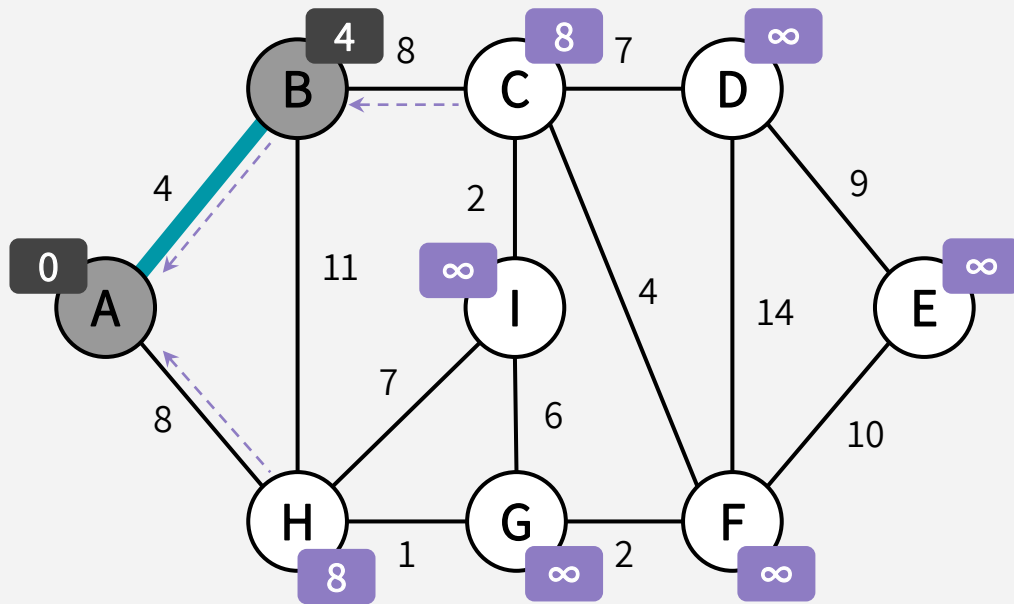
# HOW DO WE IMPLEMENT THIS?

Each vertex that's not yet reached by the growing tree keeps track of:

- 1) the **distance** from itself to the growing spanning tree using *one edge*
- 2) **how to get to there** (the closest neighbor that's reached by the tree already)

Update their estimates, and now B is officially done.

Time to choose the lightest edge on the frontier (i.e. the edge whose endpoint has the lowest distance stored)





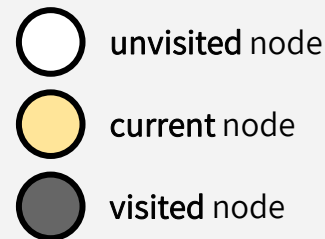
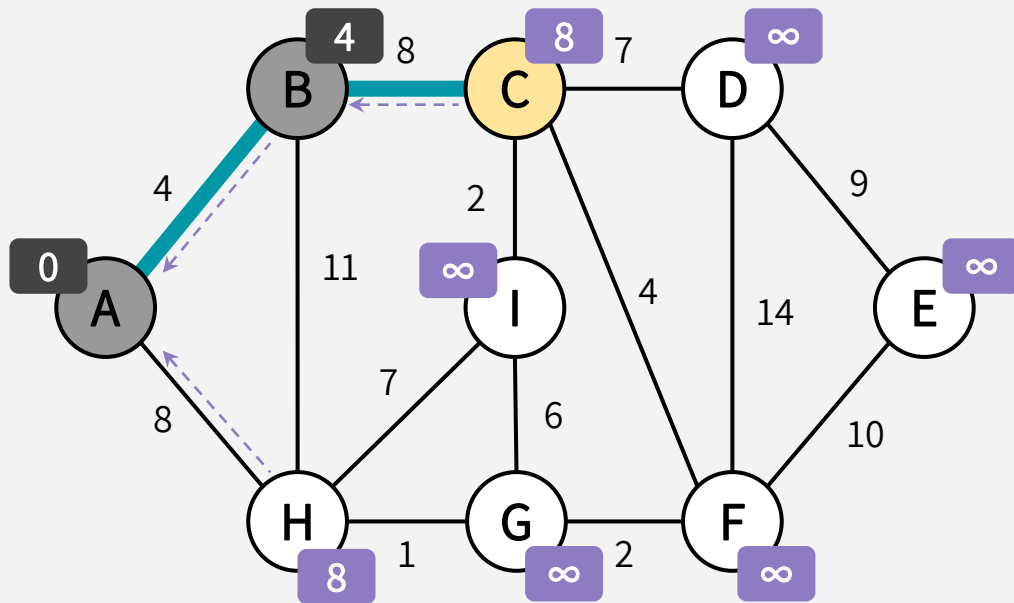
# HOW DO WE IMPLEMENT THIS?

Each vertex that's not yet reached by the growing tree keeps track of:

- 1) the **distance** from itself to the growing spanning tree using *one edge*
- 2) **how to get to there** (the closest neighbor that's reached by the tree already)

C is the closest  
node to the  
growing tree.  
(technically a tie, but let's choose C)

Since we recorded  
how to get to the tree  
from C, we know  
which edge to add.

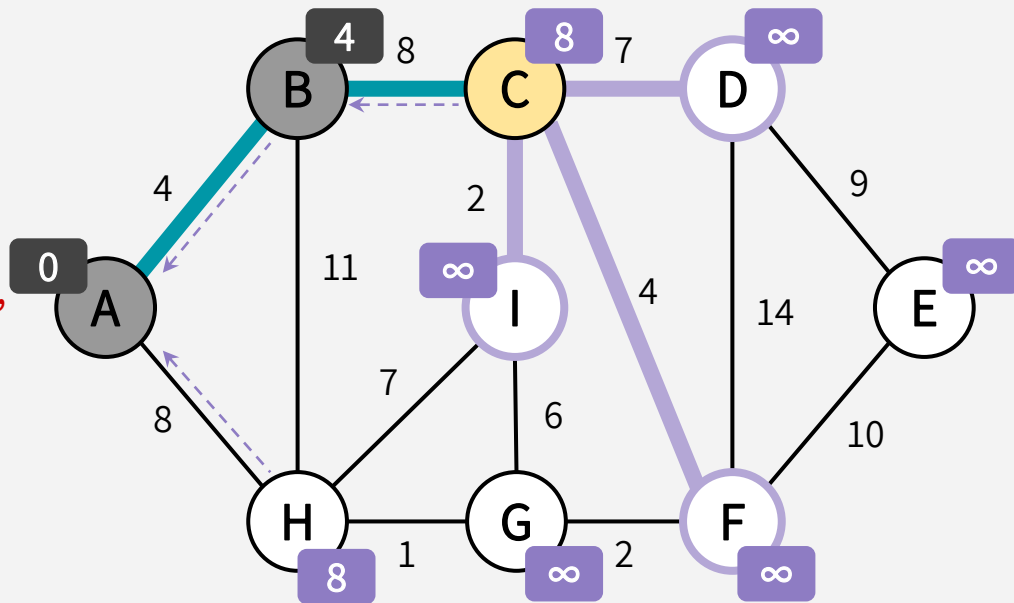


# HOW DO WE IMPLEMENT THIS?

Each vertex that's not yet reached by the growing tree keeps track of:

- 1) the **distance** from itself to the growing spanning tree using *one edge*
- 2) **how to get to there** (the closest neighbor that's reached by the tree already)

Now that C is reached by the tree, see if any of its neighbors are closer to the tree because of it!



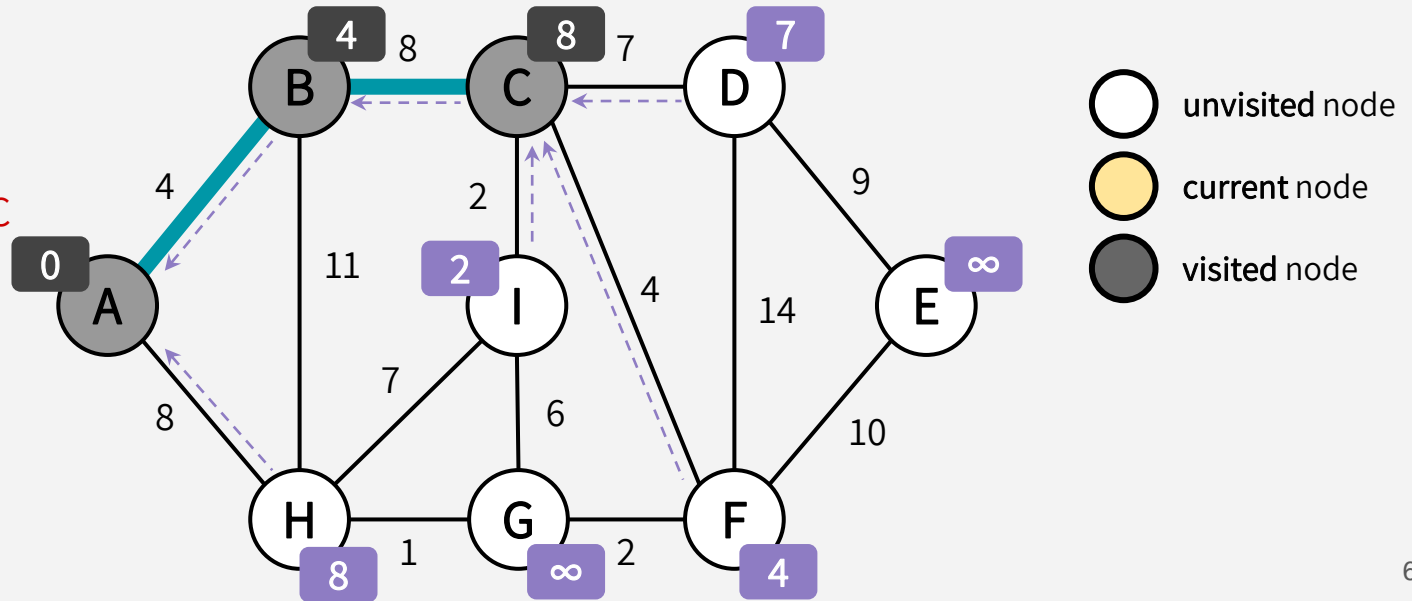
# HOW DO WE IMPLEMENT THIS?

Each vertex that's not yet reached by the growing tree keeps track of:

- 1) the **distance** from itself to the growing spanning tree using *one edge*
- 2) **how to get to there** (the closest neighbor that's reached by the tree already)

Update their estimates, and now C is officially done.

Time to choose the lightest edge on the frontier (i.e. the edge whose endpoint has the lowest distance stored)



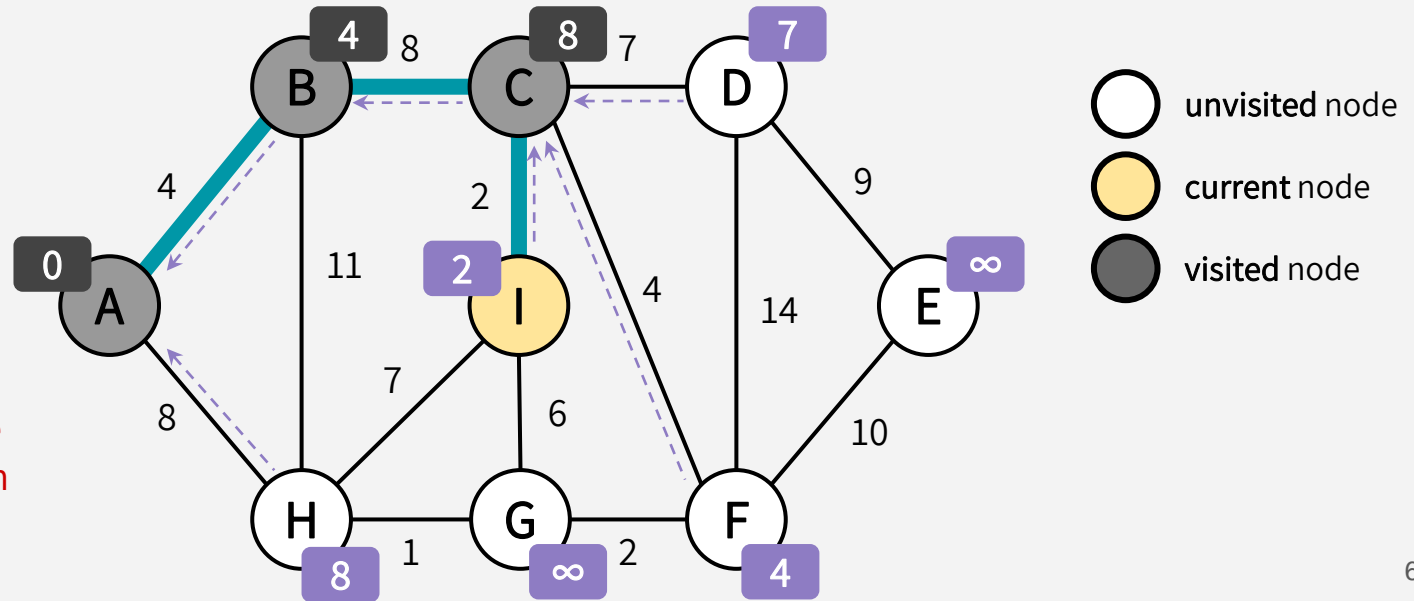
# HOW DO WE IMPLEMENT THIS?

Each vertex that's not yet reached by the growing tree keeps track of:

- 1) the **distance** from itself to the growing spanning tree using *one edge*
- 2) **how to get to there** (the closest neighbor that's reached by the tree already)

I is the closest node to the growing tree.

Since we recorded how to get to the tree from I, we know which edge to add.

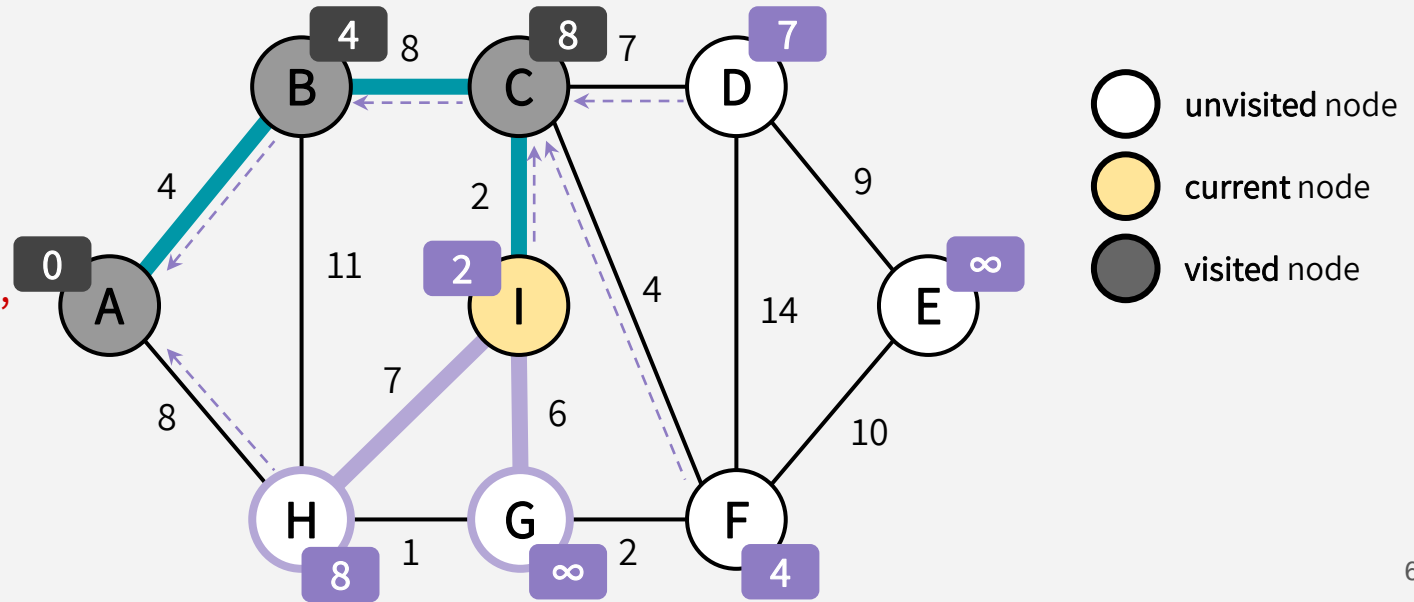


# HOW DO WE IMPLEMENT THIS?

Each vertex that's not yet reached by the growing tree keeps track of:

- 1) the **distance** from itself to the growing spanning tree using *one edge*
- 2) **how to get to there** (the closest neighbor that's reached by the tree already)

Now that I is reached by the tree, see if any of its neighbors are closer to the tree because of it!



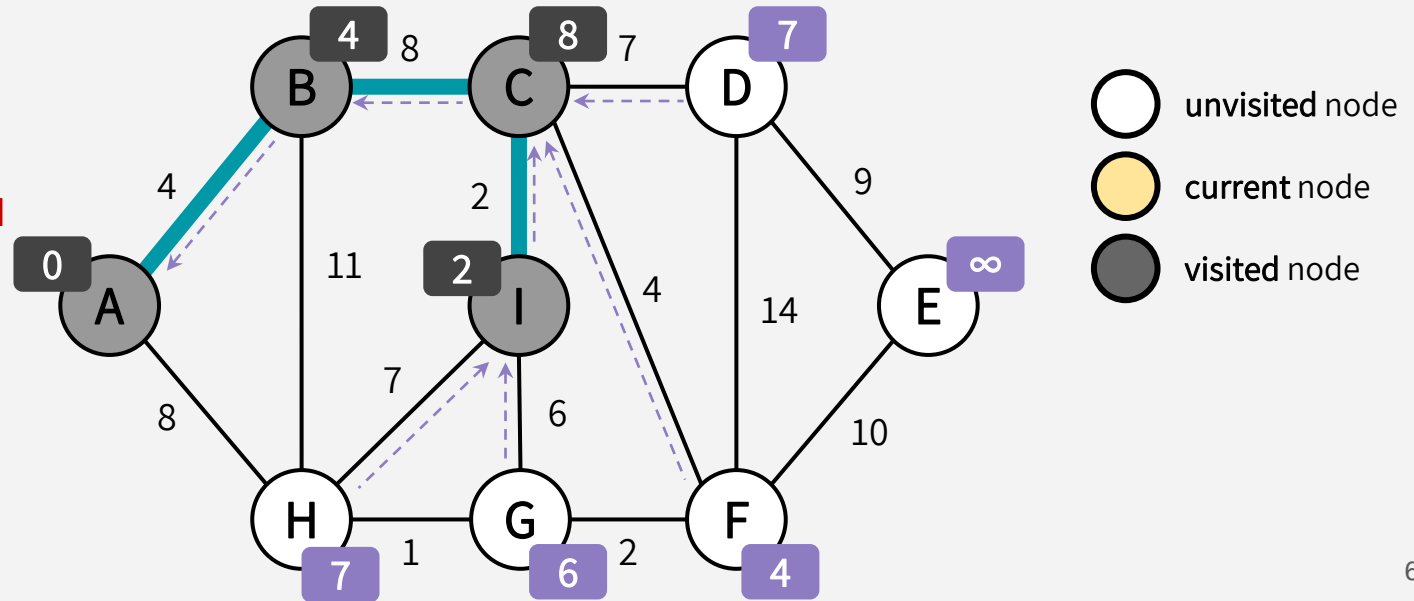
# HOW DO WE IMPLEMENT THIS?

Each vertex that's not yet reached by the growing tree keeps track of:

- 1) the **distance** from itself to the growing spanning tree using *one edge*
- 2) **how to get to there** (the closest neighbor that's reached by the tree already)

Update their estimates, and now I is officially done.

Time to choose the lightest edge on the frontier (i.e. the edge whose endpoint has the lowest distance stored)



# PRIM'S ALGORITHM: PSEUDOCODE

**PRIM**( $G = (V, E)$ ,  $s$ ):

$MST = \{\}$

$visited = \{s\}$

for all  $v$  besides  $s$ :  $d[v] = \infty$  and  $k[v] = \text{NULL}$

for each neighbor  $v$  of  $s$ :  $d[v] = w(s, v)$  and  $k[v] = s$

while  $\text{len}(visited) < n$ :

$x =$  unvisited vertex  $v$  with smallest  $d[v]$  value

$MST.add((k[x], x))$

for each unreached neighbor  $v$  of  $x$ :


$d[v] = \min(w(x, v), d[v])$

if  $d[v]$  was updated:  $k[v] = x$

$visited.add(x)$

return  $MST$

$k[v]$  stores the node in the growing tree that is closest to  $v$  (using one edge)



**Runtime (using RB-Tree):  $O(m \log n)$**

(Exact same structure as Dijkstra! We will see, Dijkstra's runtime depends on the data structure used for a priority queue.)

# PRIM'S ALGORITHM: PSEUDOCODE

**PRIM**( $G = (V, E)$ ,  $s$ ):

$MST = \{\}$

$visited = \{s\}$

for all  $v$  besides  $s$ :  $d[v] = \infty$  and  $k[v] = \text{NULL}$

for each neighbor  $v$  of  $s$ :  $d[v] = w(s, v)$  and  $k[v] = s$

while  $\text{len}(visited) < n$ :

$x =$  unvisited vertex  $v$  with smallest  $d[v]$  value

$MST.add((k[x], x))$

    for each unreached neighbor  $v$  of  $x$ :


$d[v] = \min(w(x, v), d[v])$

        if  $d[v]$  was updated:  $k[v] = x$

$visited.add(x)$

return  $MST$

$k[v]$  stores the node in the growing tree that is closest to  $v$  (using one edge)



**Runtime (using Fibonacci Heap):  $O(m + n \log n)$**

(Exact same structure as Dijkstra! We will see, Dijkstra's runtime depends on the data structure used for a priority queue.)



# PRIM'S ALGORITHM: CORRECTNESS

Let's follow our framework from before:

Prove that after each choice, you're not ruling out success.  
(i.e. you're not ruling out finding an optimal solution)

- **INDUCTIVE HYPOTHESIS:** After greedy choice  $t$ , you haven't ruled out success
- **BASE CASE:** Success is possible before you make any choices
- **INDUCTIVE STEP:** If you haven't ruled out success after choice  $t$ , then show that you won't rule out success after choice  $t+1$  (let's elaborate on this!)
- **CONCLUSION:** If you reach the end of the algorithm and haven't ruled out success then you must have succeeded

# PRIM'S ALGORITHM: CORRECTNESS

Let's follow our framework from before:

Prove that after each choice, you're not ruling out success.  
(i.e. you're not ruling out finding an optimal solution)

Our greedy choice in Prim's: **choosing the lightest edge on our frontier**  
"Not ruling out success": **there's still an MST that extends our current set of edges**

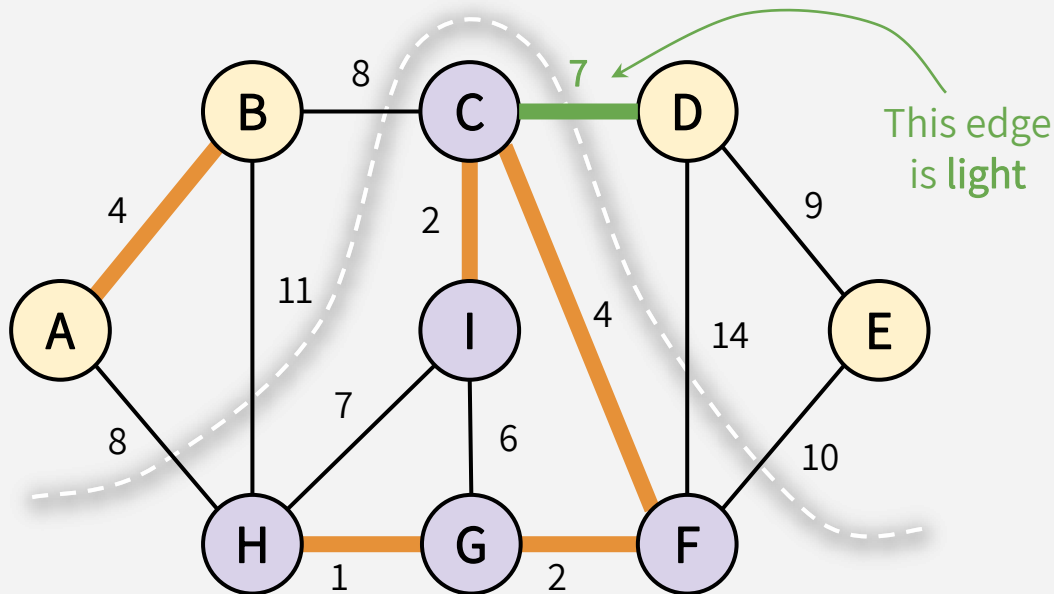
- **CONCLUSION:** If you reach the end of the algorithm and haven't ruled out success then you must have succeeded

# REMEMBER OUR LEMMA

**LEMMA:** Consider a cut that respects a set of edges  $S$ .

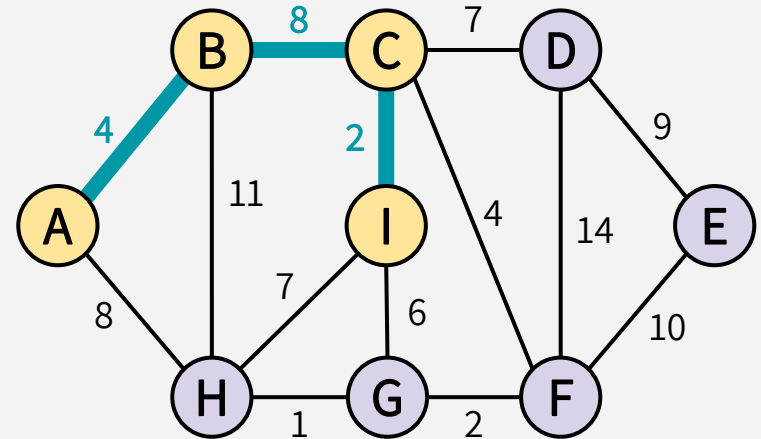
Suppose there exists an MST  $T^*$  containing  $S$ . Let  $(u,v)$  be a light edge crossing this cut.

Then, there exists an MST containing  $S \cup \{(u,v)\}$ .



# PRIM'S ALGORITHM: CORRECTNESS

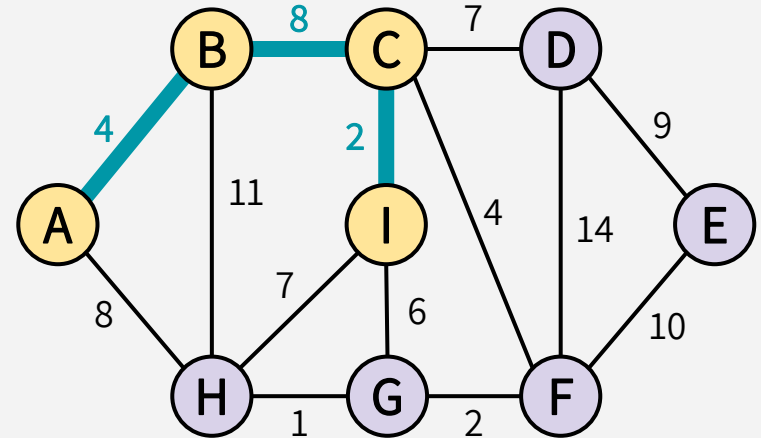
**Inductive Step (sketch):** Suppose we've already chosen a set  $S$  of  $k$  edges, and there's an MST  $T^*$  consistent with those choices. Then, Prim's chooses the *lightest edge on the frontier*, so we need to show there's an MST consistent with this new set of edges.



# PRIM'S ALGORITHM: CORRECTNESS

**Inductive Step (sketch):** Suppose we've already chosen a set  $S$  of  $k$  edges, and there's an MST  $T^*$  consistent with those choices. Then, Prim's chooses the *lightest edge on the frontier*, so we need to show there's an MST consistent with this new set of edges.

Suppose our choices  $S$  so far don't rule out success.  
This means there is an MST  $T^*$  that contains  $S$ .



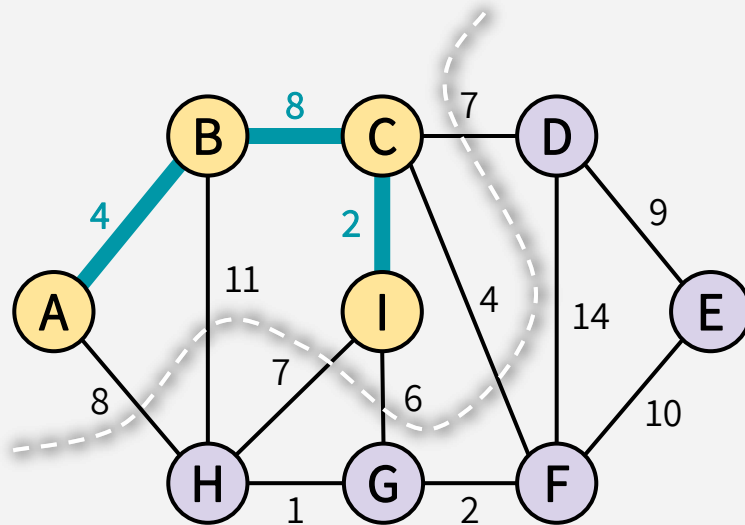
# PRIM'S ALGORITHM: CORRECTNESS

**Inductive Step (sketch):** Suppose we've already chosen a set  $S$  of  $k$  edges, and there's an MST  $T^*$  consistent with those choices. Then, Prim's chooses the *lightest edge on the frontier*, so we need to show there's an MST consistent with this new set of edges.

Suppose our choices  $S$  so far don't rule out success.  
This means there is an MST  $T^*$  that contains  $S$ .

Consider the cut {visited, unvisited}.

This cut respects the set of edges  $S$ .



# PRIM'S ALGORITHM: CORRECTNESS

**Inductive Step (sketch):** Suppose we've already chosen a set  $S$  of  $k$  edges, and there's an MST  $T^*$  consistent with those choices. Then, Prim's chooses the *lightest edge on the frontier*, so we need to show there's an MST consistent with this new set of edges.

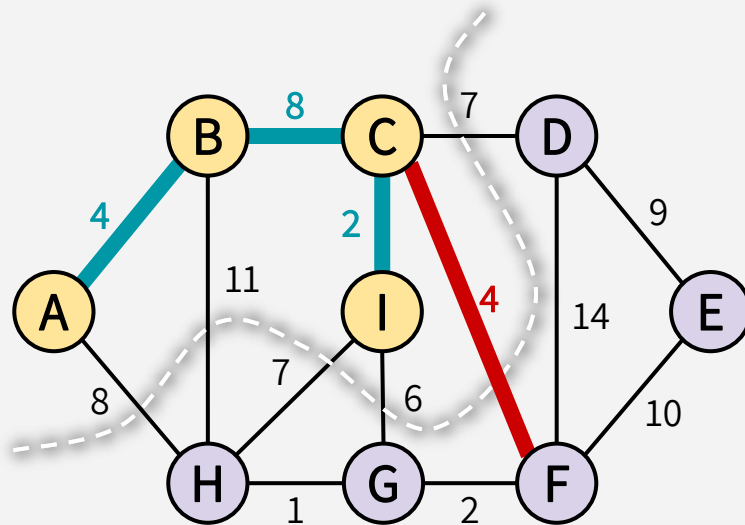
Suppose our choices  $S$  so far don't rule out success.  
This means there is an MST  $T^*$  that contains  $S$ .

Consider the cut {visited, unvisited}.

This cut respects the set of edges  $S$ .

The next edge we add is a **light edge** on this cut.

This is the smallest weight edge that crosses the cut, i.e. the *frontier* of our growing tree.



# PRIM'S ALGORITHM: CORRECTNESS

**Inductive Step (sketch):** Suppose we've already chosen a set  $S$  of  $k$  edges, and there's an MST  $T^*$  consistent with those choices. Then, Prim's chooses the *lightest edge on the frontier*, so we need to show there's an MST consistent with this new set of edges.

Suppose our choices  $S$  so far don't rule out success.  
This means there is an MST  $T^*$  that contains  $S$ .

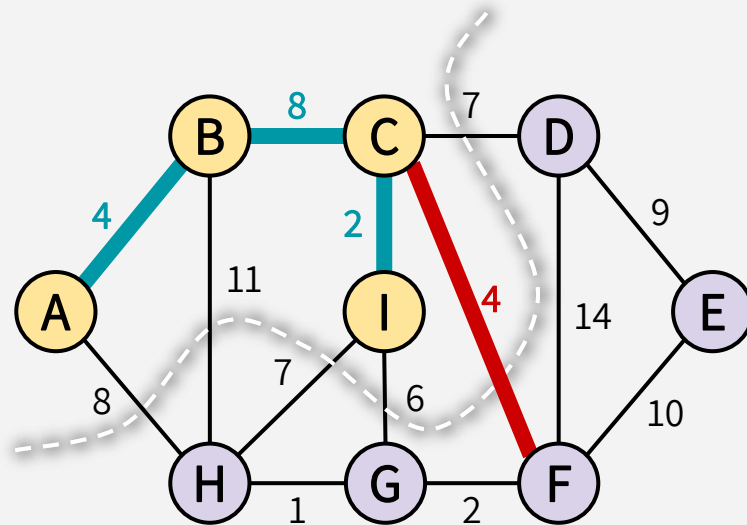
Consider the cut  $\{\text{visited}, \text{unvisited}\}$ .

This cut respects the set of edges  $S$ .

The next edge we add is a **light edge** on this cut.

This is the smallest weight edge that crosses the cut, i.e. the *frontier* of our growing tree.

By our Lemma, once we add this **light edge**, there is still an MST that is consistent with our new set of edges. Thus, we haven't ruled out success!





# PRIM'S ALGORITHM: CORRECTNESS

## INDUCTIVE HYPOTHESIS

After adding the  $t^{\text{th}}$  edge, there is an MST that contains the edges added so far.

## BASE CASE

After adding the  $0^{\text{th}}$  edge, there exists an MST with the edges added so far.

## INDUCTIVE STEP (*weak induction*)

If the inductive hypothesis holds for  $t$  (i.e. the edge choices so far are safe), then it holds for  $t+1$ , as there is still an MST that contains these  $t+1$  edges. **We proved this by considering the cut between visited & unvisited nodes (i.e. the “frontier”) and invoking our Lemma from earlier in class.**

## CONCLUSION

After adding the  $(n-1)^{\text{st}}$  edge, there exists an MST containing the edges added so far. A tree containing  $n-1$  edges is already a spanning tree, so the tree we have must be a minimum spanning tree.

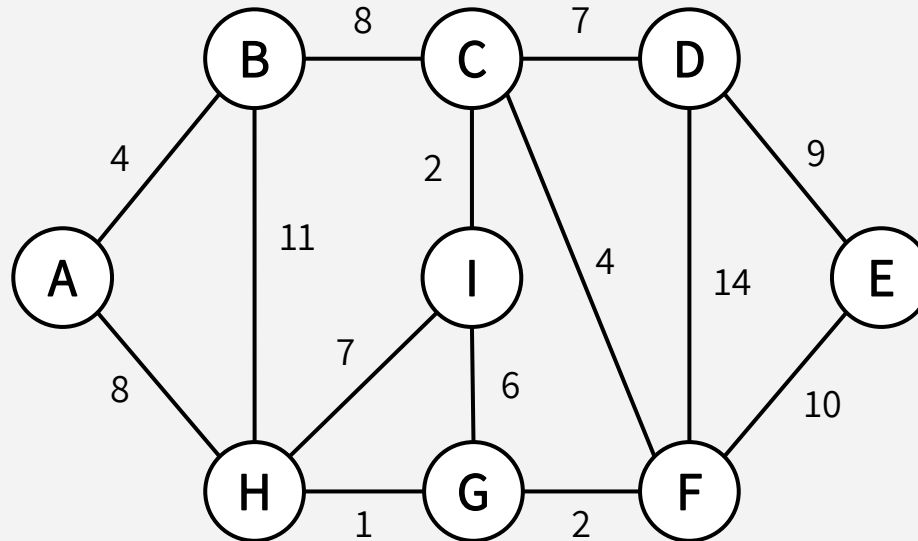
# KRUSKAL'S ALGORITHM

Greedily add the cheapest edge!

# KRUSKAL'S ALGORITHM: THE IDEA

## Greedy choice:

Maintain a forest of trees, & greedily add the cheapest edge to combine trees

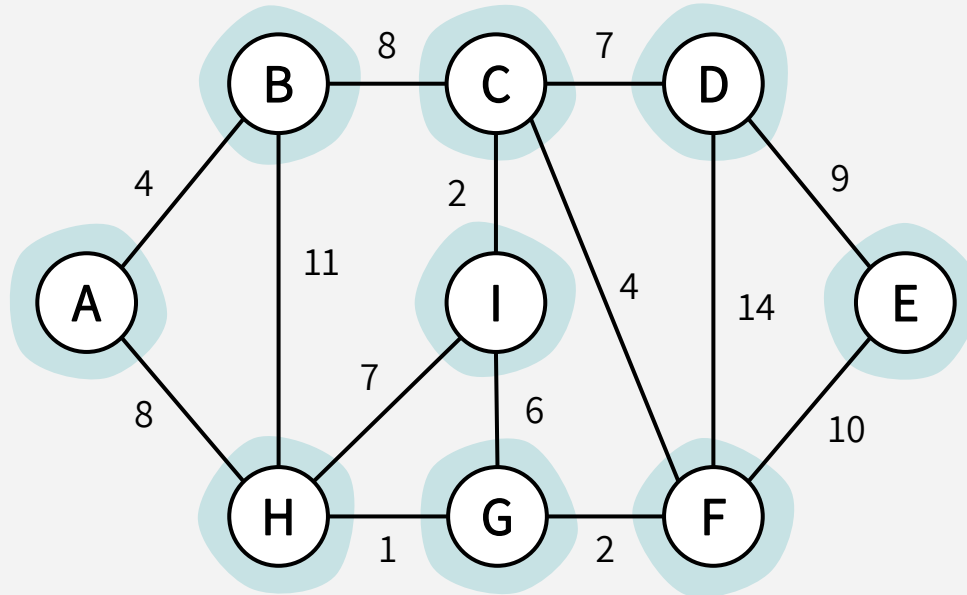


# KRUSKAL'S ALGORITHM: THE IDEA

## Greedy choice:

Maintain a forest of trees, & greedily add the cheapest edge to combine trees

Every node on its  
own starts as an  
individual tree in  
this forest

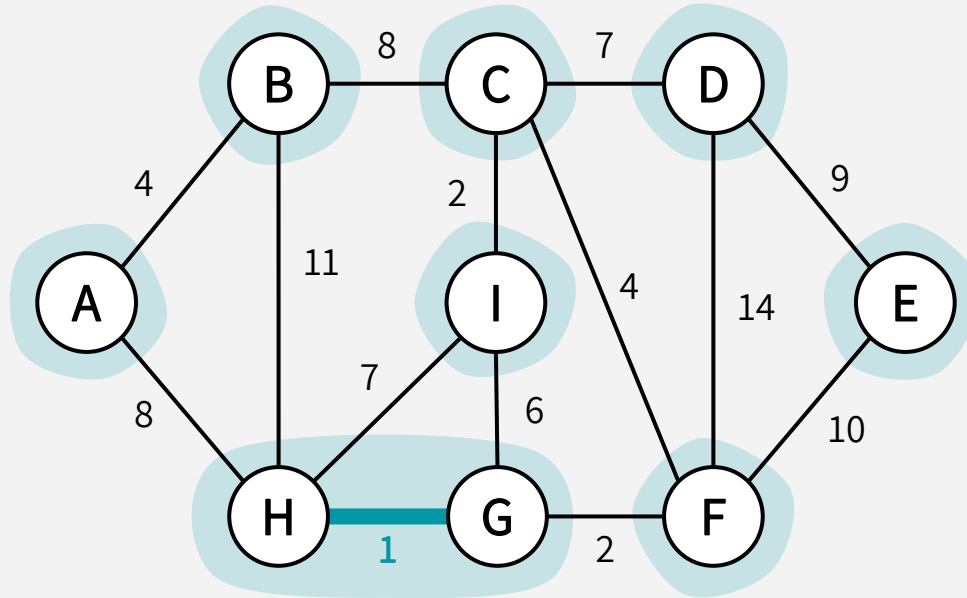


# KRUSKAL'S ALGORITHM: THE IDEA

## Greedy choice:

Maintain a forest of trees, & greedily add the cheapest edge to combine trees

Choose the  
cheapest edge that  
would combine  
two trees  
(i.e. that won't cause a cycle)

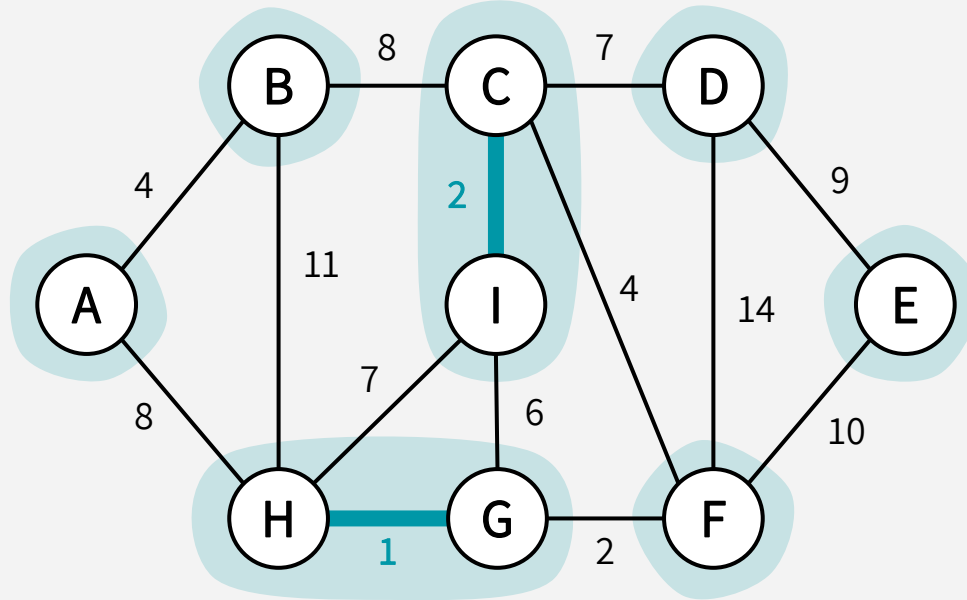


# KRUSKAL'S ALGORITHM: THE IDEA

## Greedy choice:

Maintain a forest of trees, & greedily add the cheapest edge to combine trees

Choose the  
cheapest edge that  
would combine  
two trees  
(i.e. that won't cause a cycle)



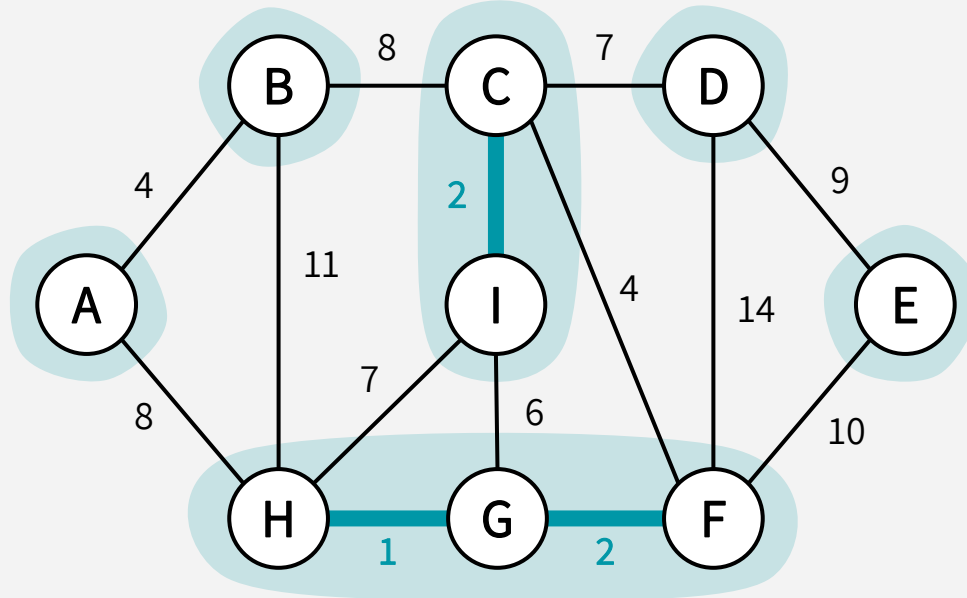
If there's a tie, choose  
one of the edges

# KRUSKAL'S ALGORITHM: THE IDEA

## Greedy choice:

Maintain a forest of trees, & greedily add the cheapest edge to combine trees

Choose the  
cheapest edge that  
would combine  
two trees  
(i.e. that won't cause a cycle)

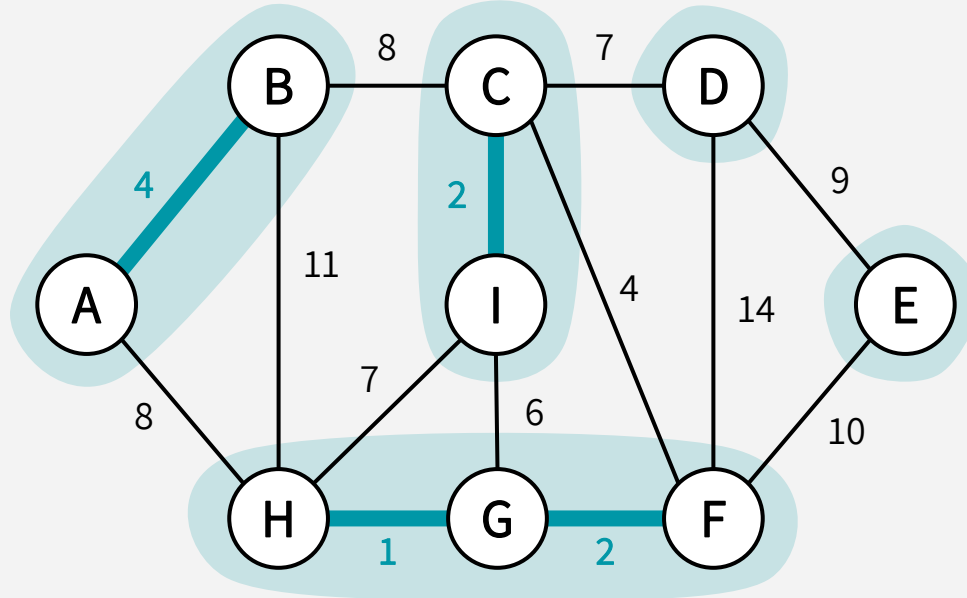


# KRUSKAL'S ALGORITHM: THE IDEA

## Greedy choice:

Maintain a forest of trees, & greedily add the cheapest edge to combine trees

Choose the  
cheapest edge that  
would combine  
two trees  
(i.e. that won't cause a cycle)



If there's a tie, choose  
one of the edges

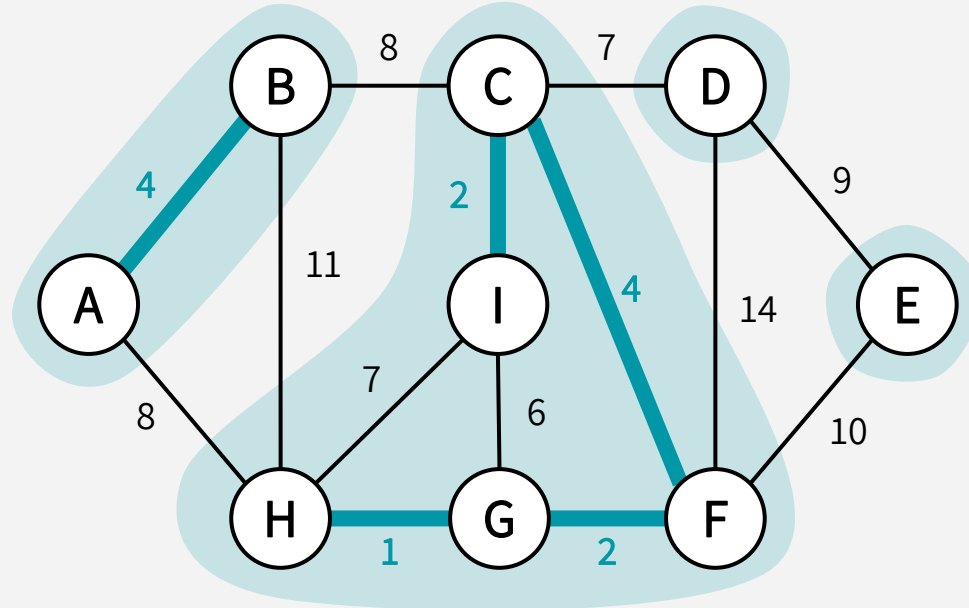


# KRUSKAL'S ALGORITHM: THE IDEA

## Greedy choice:

Maintain a forest of trees, & greedily add the cheapest edge to combine trees

Choose the  
cheapest edge that  
would combine  
two trees  
(i.e. that won't cause a cycle)

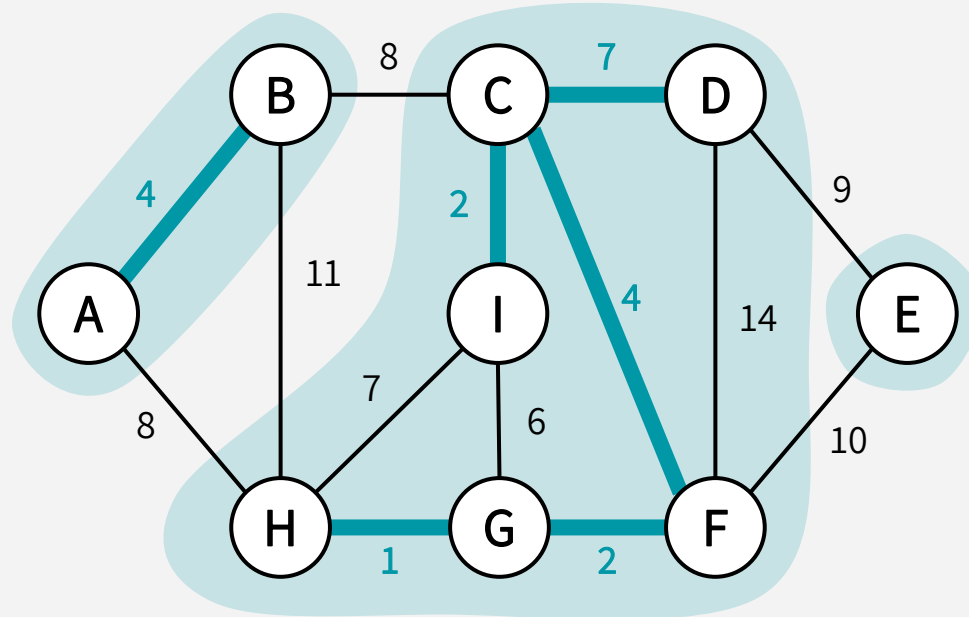


# KRUSKAL'S ALGORITHM: THE IDEA

## Greedy choice:

Maintain a forest of trees, & greedily add the cheapest edge to combine trees

Choose the  
cheapest edge that  
would combine  
two trees  
(i.e. that won't cause a cycle)

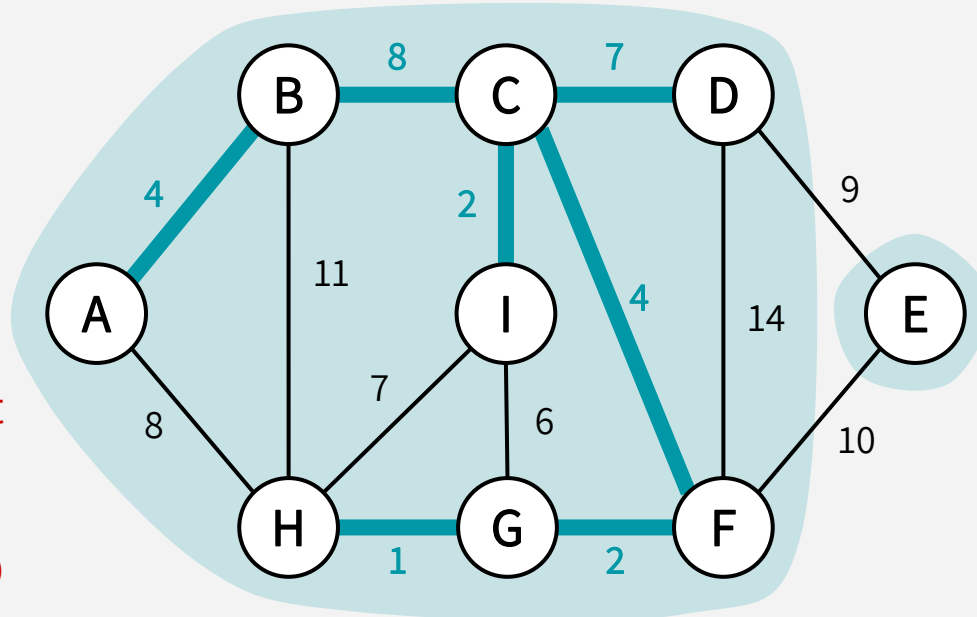


# KRUSKAL'S ALGORITHM: THE IDEA

Greedy choice:

Maintain a forest of trees, & greedily add the cheapest edge to combine trees

Choose the  
cheapest edge that  
would combine  
two trees  
(i.e. that won't cause a cycle)

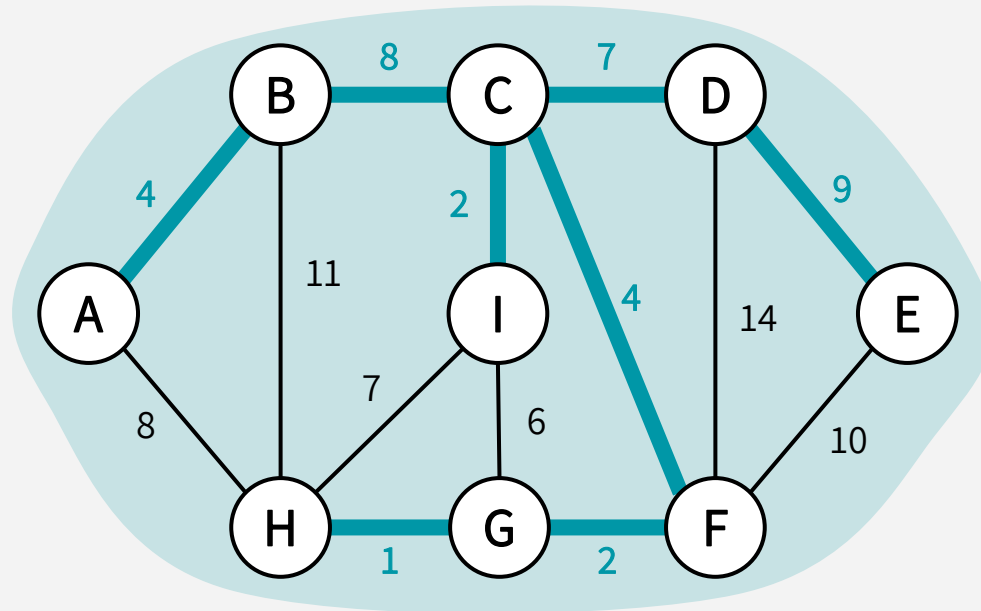


# KRUSKAL'S ALGORITHM: THE IDEA

## Greedy choice:

Maintain a forest of trees, & greedily add the cheapest edge to combine trees

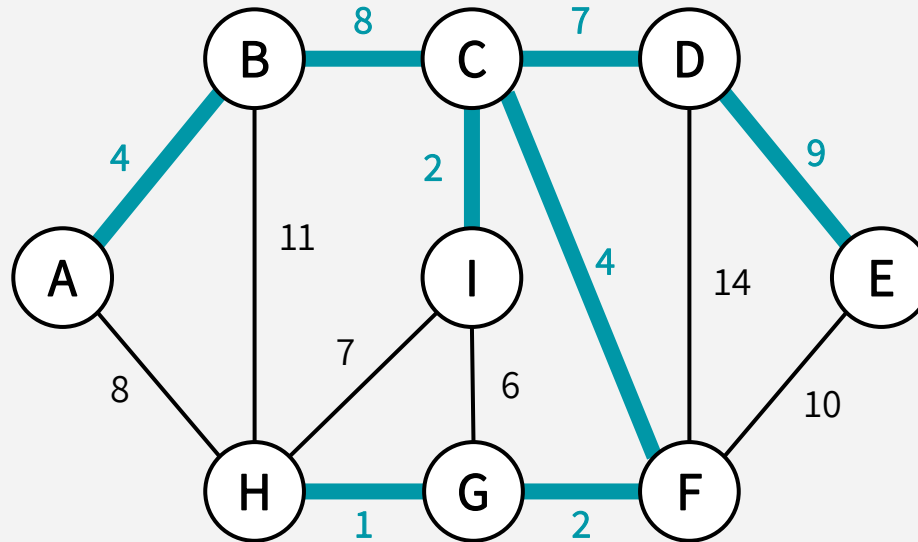
Choose the  
cheapest edge that  
would combine  
two trees  
(i.e. that won't cause a cycle)



# KRUSKAL'S ALGORITHM: THE IDEA

**Greedy choice:**

Maintain a forest of trees, & greedily add the cheapest edge to combine trees



We're done!  
This is the MST.

# KRUSKAL'S ALGORITHM: PSEUDOCODE

```
KRUSKAL_NOT_VERY_DETAILED( $G = (V, E)$ ):
```

```
     $E\_SORTED = E$  sorted by weight in non-decreasing order
```

```
     $MST = \{\}$ 
```

```
    for  $v$  in  $V$ :
```

```
        put  $v$  in its own tree
```

```
    for  $(u, v)$  in  $E\_SORTED$ :
```

```
        if  $u$ 's tree and  $v$ 's tree are not the same:
```

```
             $MST.add((u, v))$ 
```

```
            merge  $u$ 's tree with  $v$ 's tree
```

```
    return  $MST$ 
```

# KRUSKAL'S ALGORITHM: PSEUDOCODE

```
KRUSKAL_NOT_VERY_DETAILED( $G = (V, E)$ ):  
     $E\_SORTED = E$  sorted by weight in non-decreasing order  
     $MST = \{\}$   
    for  $v$  in  $V$ :  
        put  $v$  in its own tree  
    for  $(u, v)$  in  $E\_SORTED$ :  
        if  $u$ 's tree and  $v$ 's tree are not the same:  
             $MST.add((u, v))$   
            merge  $u$ 's tree with  $v$ 's tree  
    return  $MST$ 
```

To implement these lines, we'll use a *Union-Find data structure*, which supports 3 operations: **MAKE\_SET(x)**, **FIND(x)**, and **UNION(x, y)**

# KRUSKAL'S ALGORITHM: PSEUDOCODE

**MAKE\_SET(x)**: creates a set {x} in  $O(1)$

**FIND(x)**: returns the set containing x in  $O(1)$

**UNION(x, y)**: merges the sets containing x and y in  $O(1)$

To implement these lines, we'll use a *Union-Find data structure*, which supports 3 operations: **MAKE\_SET(x)**, **FIND(x)**, and **UNION(x, y)**



# KRUSKAL'S ALGORITHM: PSEUDOCODE

```
KRUSKAL( $G = (V, E)$ ):
```

```
   $E\_SORTED = E$  sorted by weight in non-decreasing order
```

```
   $MST = \{\}$ 
```

```
  for  $v$  in  $V$ :
```

```
    MAKE_SET( $v$ )
```

```
  for  $(u, v)$  in  $E\_SORTED$ :
```


```
    if FIND( $u$ )  $\neq$  FIND( $v$ ):
```

```
       $MST.add((u, v))$ 
```

```
      UNION( $u, v$ )
```

```
  return  $MST$ 
```

Basically, the time to sort the edge weights dominates the runtime.  
 $O(m \log m) = O(m \log n)$ , since  $m \leq n^2$



(With union-find data structure) **Runtime =  $O(m \log n)$**

# KRUSKAL'S ALGORITHM: PSEUDOCODE

```
KRUSKAL(G = (V,E)):
```

```
    E_SORTED = E sorted by weight in non-decreasing order
```

```
    MST = {}
```

```
    for v in V:
```

```
        MAKE_SET(v)
```

```
    for (u,v) in E_SORTED:
```


```
        if FIND(u) != FIND(v):
```

```
            MST.add((u,v))
```

```
            UNION(u,v)
```

```
    return MST
```

If the edge weights are of appropriate values and RadixSort can be applied instead



(With union-find data structure & RadixSort) **Runtime =  $O(m)$**

# KRUSKAL'S CORRECTNESS

Let's follow our framework from before:

Prove that after each choice, you're not ruling out success.  
(i.e. you're not ruling out finding an optimal solution)

- **INDUCTIVE HYPOTHESIS:** After greedy choice  $t$ , you haven't ruled out success
- **BASE CASE:** Success is possible before you make any choices
- **INDUCTIVE STEP:** If you haven't ruled out success after choice  $t$ , then show that you won't rule out success after choice  $t+1$  (let's elaborate on this!)
- **CONCLUSION:** If you reach the end of the algorithm and haven't ruled out success then you must have succeeded

# KRUSKAL'S CORRECTNESS

Let's follow our framework from before:

Prove that after each choice, you're not ruling out success.  
(i.e. you're not ruling out finding an optimal solution)

Our greedy choice: **choosing the cheapest edge that combines two trees**

"Not ruling out success": **there's still an MST that extends our current set of edges**

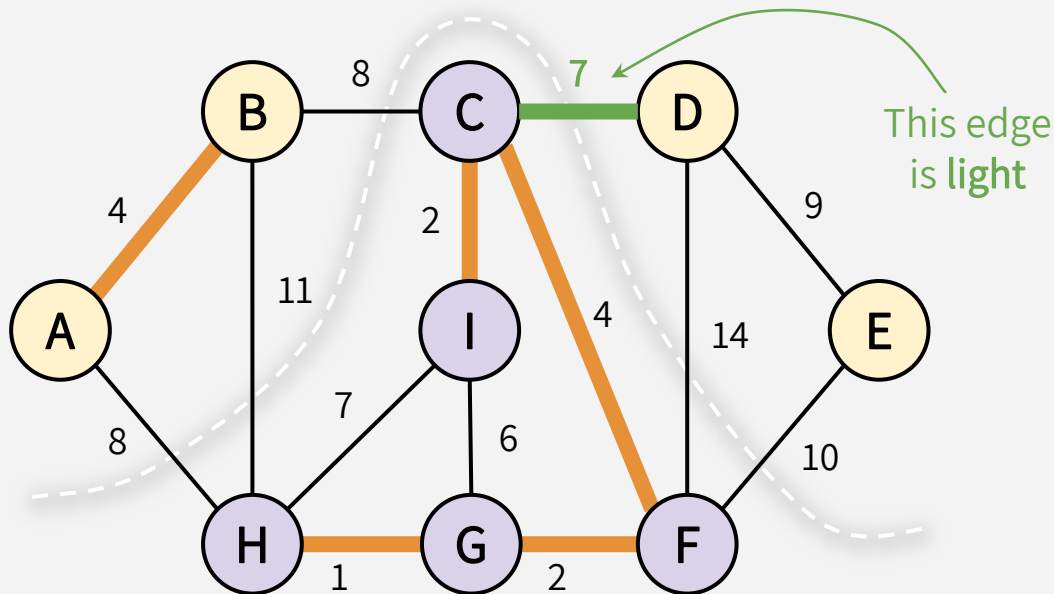
- **CONCLUSION:** If you reach the end of the algorithm and haven't ruled out success then you must have succeeded

# REMEMBER OUR LEMMA

**LEMMA:** Consider a cut that respects a set of edges  $S$ .

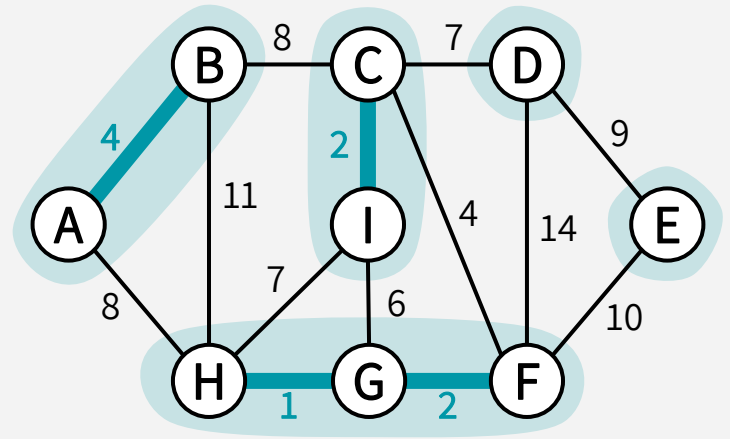
Suppose there exists an MST  $T^*$  containing  $S$ . Let  $(u,v)$  be a light edge crossing this cut.

Then, there exists an MST containing  $S \cup \{(u,v)\}$ .



# KRUSKAL'S CORRECTNESS

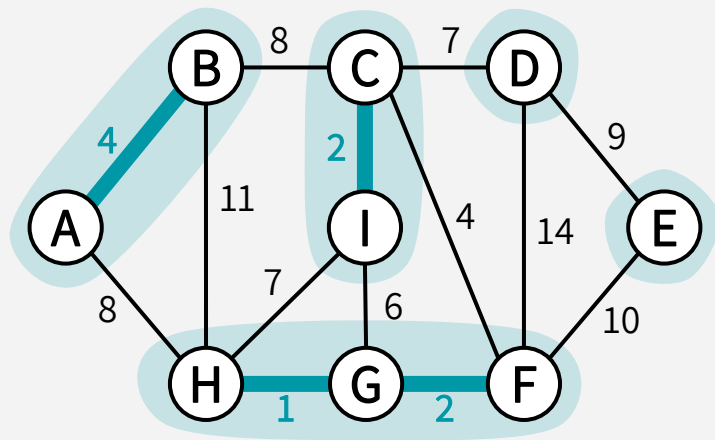
**Inductive Step (sketch):** Suppose we've already chosen a set  $S$  of  $k$  edges, and there's an MST  $T^*$  consistent with those choices. Then, Kruskal's adds the *cheapest edge that would merge 2 trees*, so we show there's still an MST consistent with this new set of edges.



# KRUSKAL'S CORRECTNESS

**Inductive Step (sketch):** Suppose we've already chosen a set  $S$  of  $k$  edges, and there's an MST  $T^*$  consistent with those choices. Then, Kruskal's adds the *cheapest edge that would merge 2 trees*, so we show there's still an MST consistent with this new set of edges.

Suppose our choices  $S$  so far don't rule out success.  
This means there is an MST  $T^*$  that contains  $S$ .

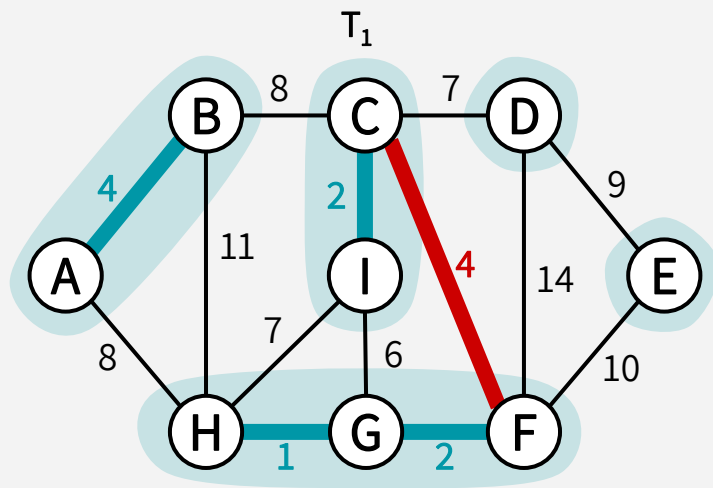


# KRUSKAL'S CORRECTNESS

**Inductive Step (sketch):** Suppose we've already chosen a set  $S$  of  $k$  edges, and there's an MST  $T^*$  consistent with those choices. Then, Kruskal's adds the *cheapest edge that would merge 2 trees*, so we show there's still an MST consistent with this new set of edges.

Suppose our choices  $S$  so far don't rule out success.  
This means there is an MST  $T^*$  that contains  $S$ .

The **next edge** we add will merge two trees,  $T_1$  &  $T_2$   
This edge is the cheapest edge that bridge two trees.





# KRUSKAL'S CORRECTNESS

**Inductive Step (sketch):** Suppose we've already chosen a set  $S$  of  $k$  edges, and there's an MST  $T^*$  consistent with those choices. Then, Kruskal's adds the *cheapest edge that would merge 2 trees*, so we show there's still an MST consistent with this new set of edges.

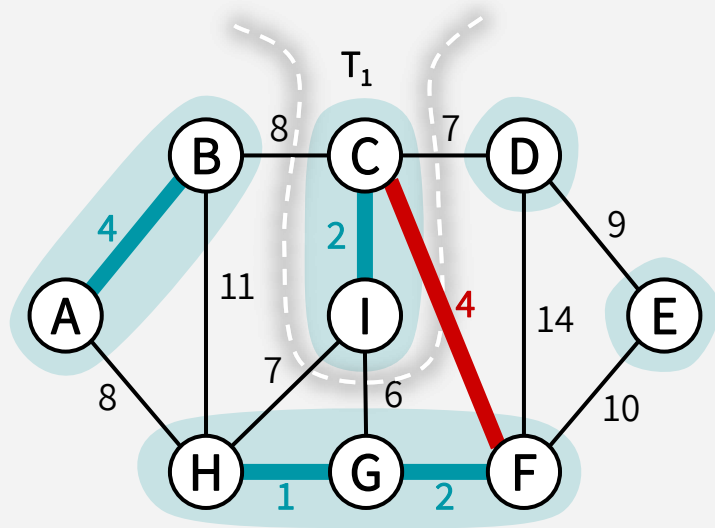
Suppose our choices  $S$  so far don't rule out success. This means there is an MST  $T^*$  that contains  $S$ .

The **next edge** we add will merge two trees,  $T_1$  &  $T_2$

This edge is the cheapest edge that bridge two trees.

Consider the cut  $\{T_1, V - T_1\}$ .

This cut respects  $S$ . Our new edge is *light* for the cut (it's the cheapest edge after all).



# KRUSKAL'S CORRECTNESS

**Inductive Step (sketch):** Suppose we've already chosen a set  $S$  of  $k$  edges, and there's an MST  $T^*$  consistent with those choices. Then, Kruskal's adds the *cheapest edge that would merge 2 trees*, so we show there's still an MST consistent with this new set of edges.

Suppose our choices  $S$  so far don't rule out success. This means there is an MST  $T^*$  that contains  $S$ .

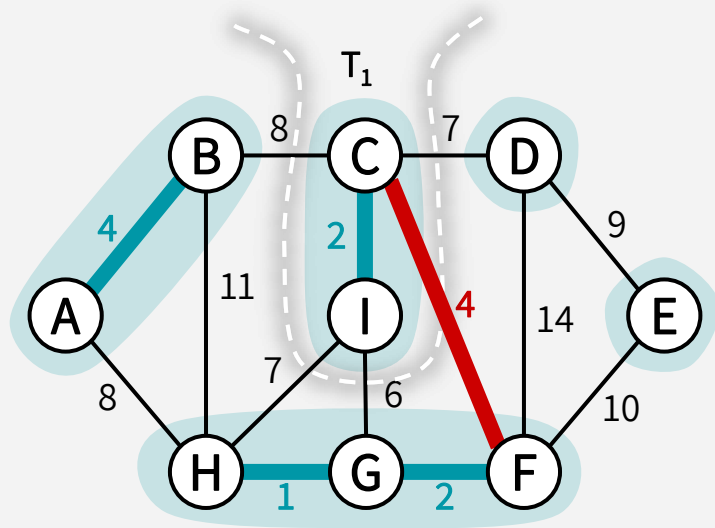
The **next edge** we add will merge two trees,  $T_1$  &  $T_2$

This edge is the cheapest edge that bridge two trees.

Consider the cut  $\{T_1, V - T_1\}$ .

This cut respects  $S$ . Our new edge is *light* for the cut (it's the cheapest edge after all).

By our Lemma, once we add this **light edge**, there is still an MST that is consistent with our new set of edges. Thus, we haven't ruled out success!



# KRUSKAL'S CORRECTNESS

## INDUCTIVE HYPOTHESIS

After adding the  $t^{\text{th}}$  edge, there is an MST that contains the edges added so far.

## BASE CASE

After adding the  $0^{\text{th}}$  edge, there exists an MST with the edges added so far.

## INDUCTIVE STEP (*weak induction*)

If the inductive hypothesis holds for  $t$  (i.e. the edge choices so far are safe), then it holds for  $t+1$ , as there is still an MST that contains these  $t+1$  edges. **We proved this by considering the cut between the tree living at one endpoint of our chosen edge & all remaining vertices, & invoking our favorite Lemma.**

## CONCLUSION

After adding the  $(n-1)^{\text{st}}$  edge, there exists an MST containing the edges added so far. A tree containing  $n-1$  edges is already a spanning tree, so the tree we have must be a minimum spanning tree.

# PRIM'S vs. KRUSKAL'S

## Prim's Algorithm

Grows a single tree by greedily adding the cheapest edge on the “frontier” of the growing tree.

Runtime (RB-tree):  $O(m \log n)$

Runtime (Fibonacci Heap):  $O(m + n \log n)$

Prim's may be better on dense graphs (where  $m$  is  $\sim n^2$ ) if you can't RadixSort edge weights

## Kruskal's Algorithm

Maintains a forest and greedily chooses the cheapest edge that would be able to merge two trees

Runtime (union-find data struct.):  $O(m \log n)$

Runtime (union-find + radixSort) :  $O(m)$

Kruskal's may be better on sparse graphs if you *can* RadixSort edge weights

**Both are greedy algorithms, with similar reasoning (that piggyback off of our lemma).**

Optimal substructure: subgraphs generated by cuts — the way to make safe choices is to choose light edges crossing the cut.

# CAN WE DO BETTER?

The algorithms are all  
comparison-based!

Karger-Klein Tarjan (1995)

$O(m)$  expected time *randomized* algorithm

Chazelle (2000)

$O(m \cdot \alpha(n))$  time *deterministic* algorithm

Pettie-Ramachandran (2002)

$O\left(\begin{array}{l} \text{optimal \# of comparisons...} \\ \text{whatever that is (i.e. if there exists} \\ \text{an algo which uses } X \text{ comparisons,} \\ \text{this algo will run in time } O(X) \end{array}\right)$  time deterministic algorithm

↖ This bound is unknown!  
For now, we know it's  $\Omega(n)$  and  $O(m \cdot \alpha(n))$ .

# Acknowledgement

- Stanford University