MINIMUM SPANNING TREES

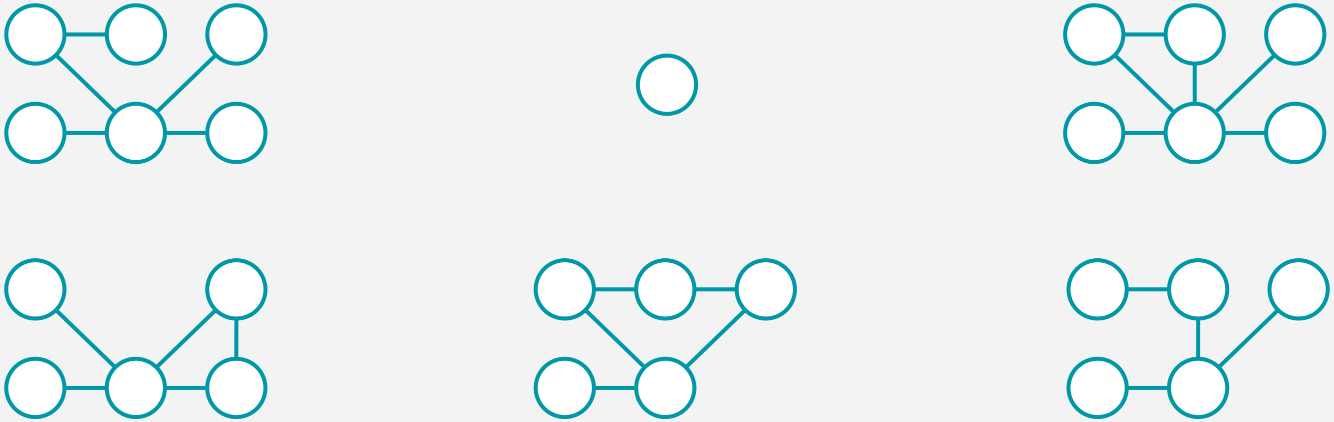
What are minimum spanning trees (MSTs)?

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?

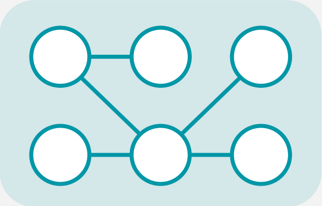


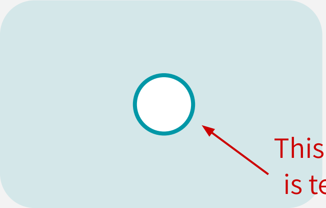
TREES IN GRAPHS

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?



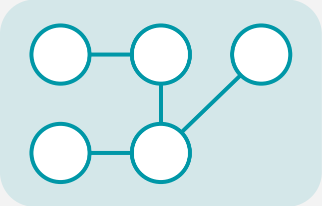


This single node is technically a valid tree!

Contains cycle

Contains cycle

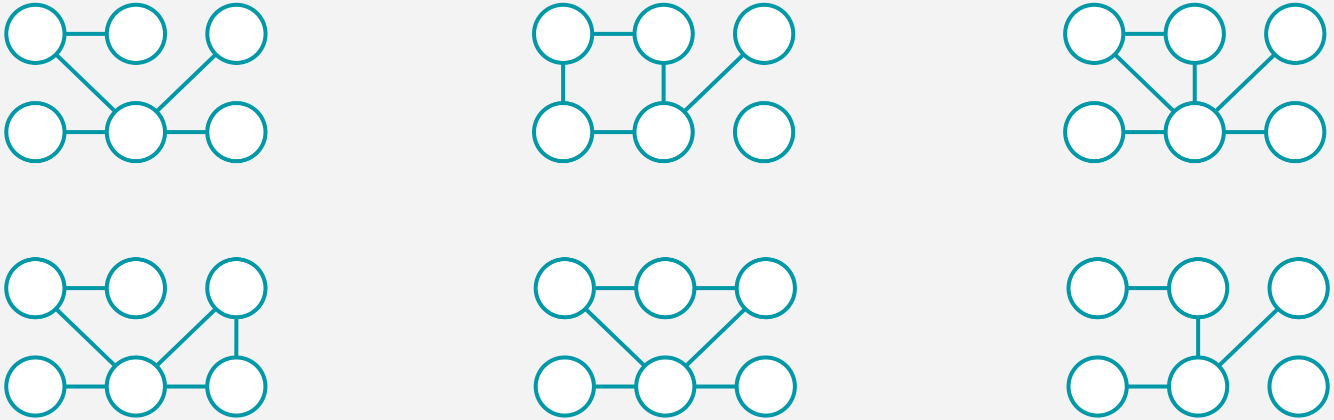
Contains cycle



SPANNING TREES

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

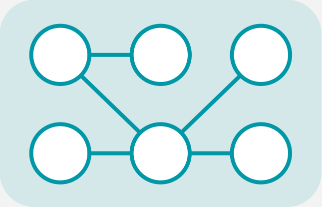
Which of these are spanning trees?



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

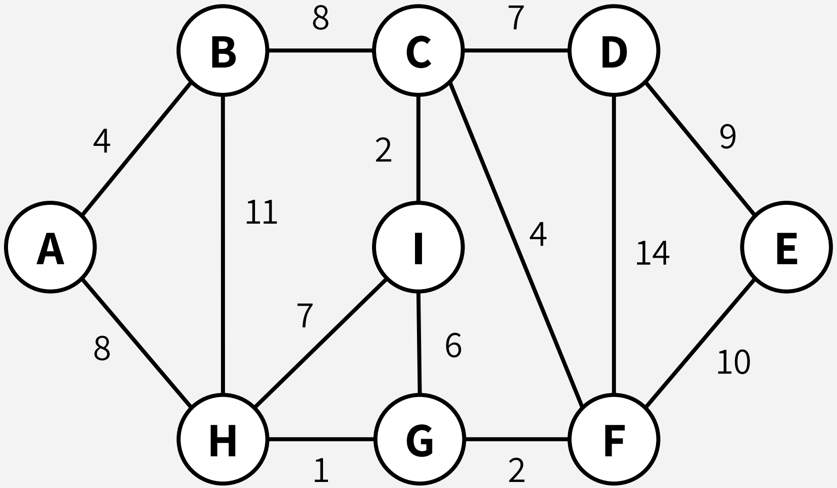
Doesn’t connect all vertices

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.



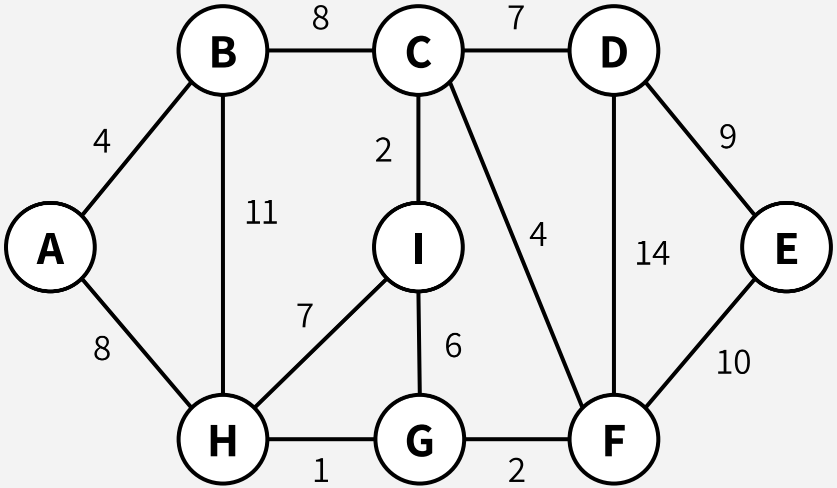
MINIMUM SPANNING TREES (MSTs)

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)



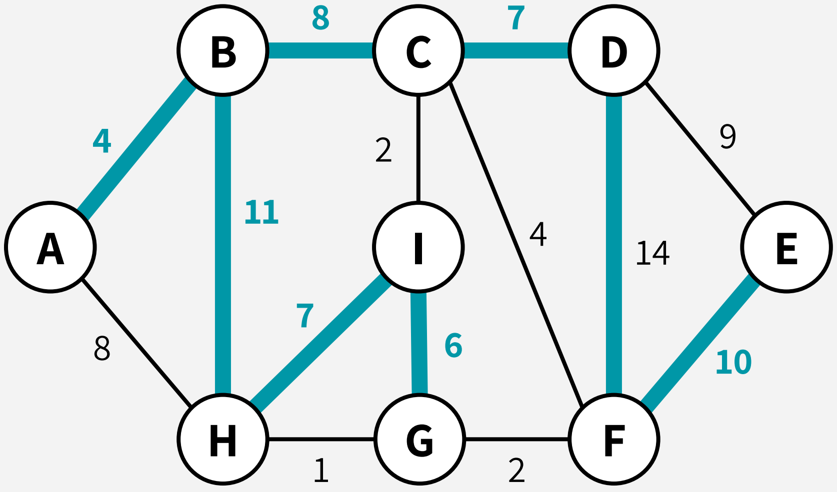
MINIMUM SPANNING TREES (MSTs)

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**.

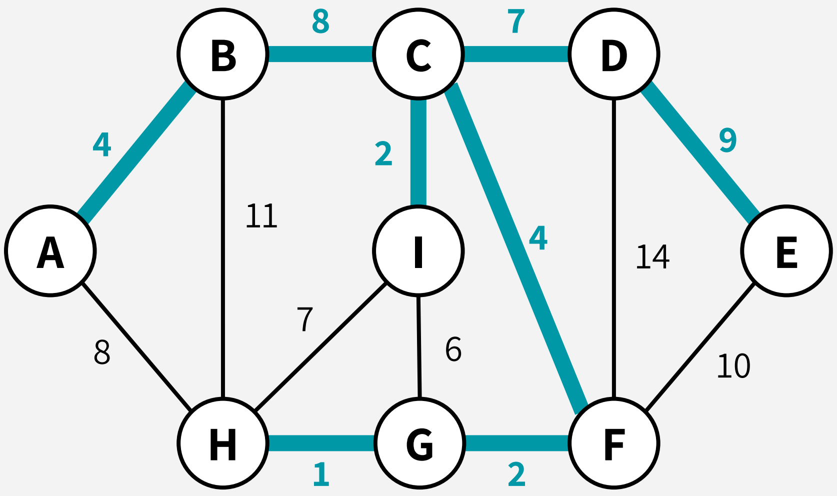
MINIMUM SPANNING TREES (MSTs)

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)

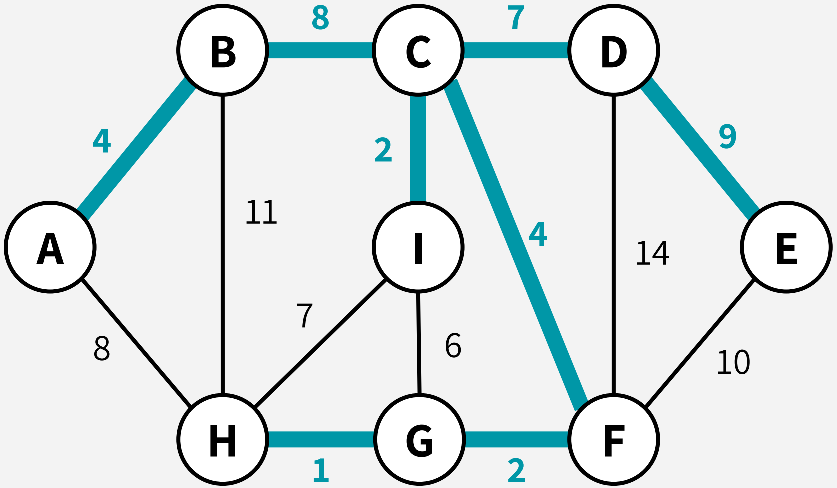


This spanning tree has a cost of **37**.

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

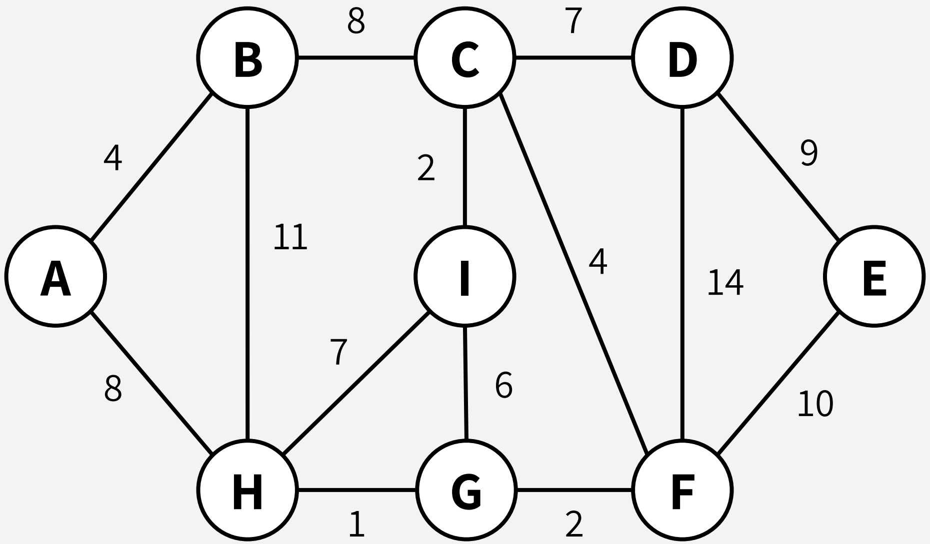
Find clusters in a dataset (one of the algorithms we'll see today can be modified slightly to basically do this)

Useful primitive

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

CUTS IN GRAPHS

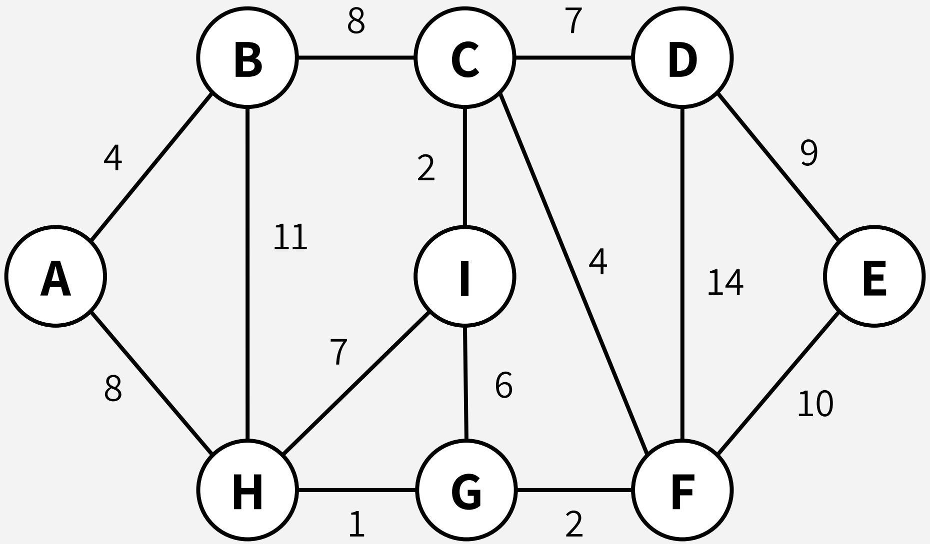
A **cut** is a partition of the vertices into two nonempty parts.



PRIM'S ALGORITHM: THE IDEA

Greedy choice:

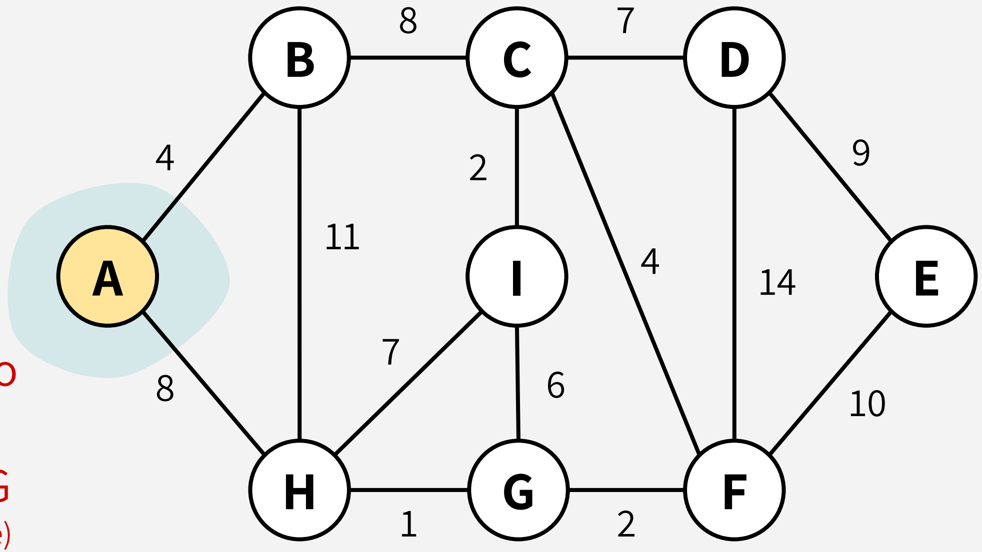
Grow a single tree, & greedily add the shortest edge that could grow our tree



PRIM'S ALGORITHM: THE IDEA

Greedy choice:

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



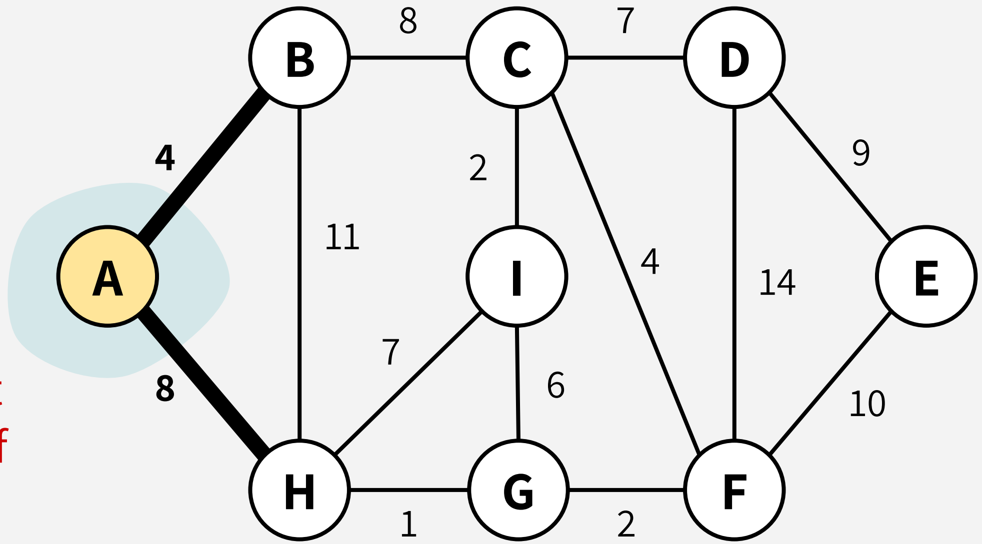
First, we can initialize our tree to contain a single arbitrary node in G

(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

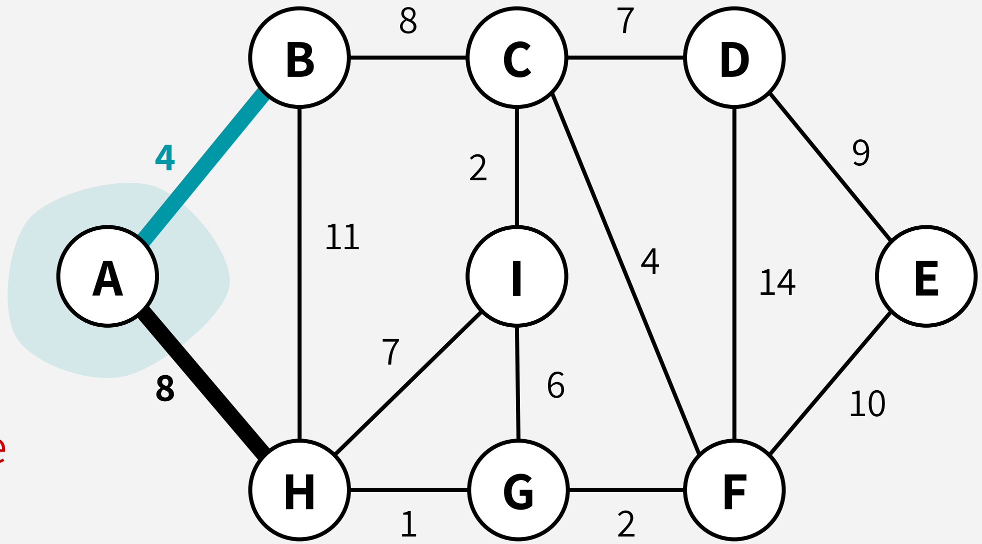


Consider the edges coming out of the "frontier" of our growing tree.

PRIM'S ALGORITHM: THE IDEA

Greedy choice:

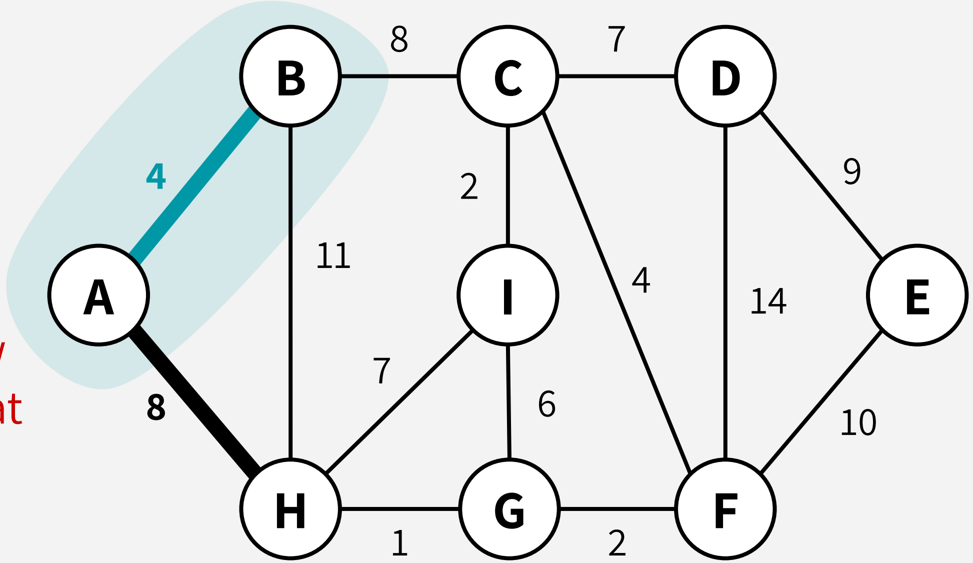
Grow a single tree, & greedily add the shortest edge that could grow our tree



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

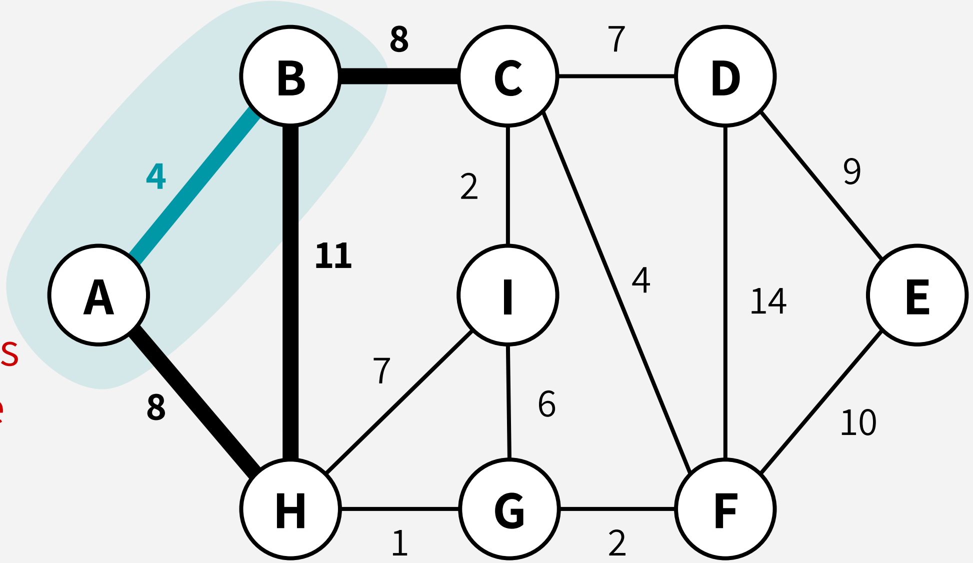


Our tree just grew by one! Now repeat until we reach all the nodes.

PRIM'S ALGORITHM: THE IDEA

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

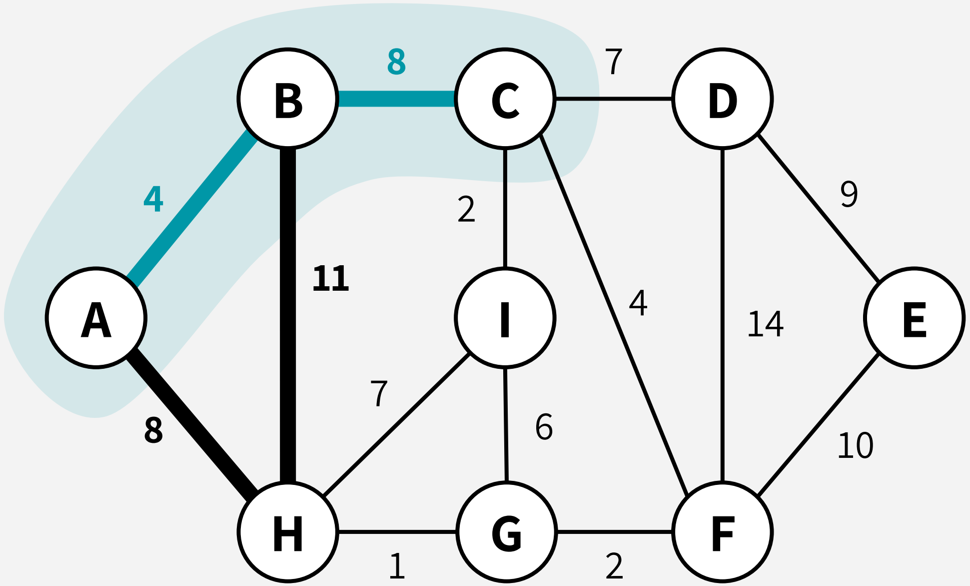
Consider the edges coming out of the "frontier" of our growing tree.



PRIM'S ALGORITHM: THE IDEA

Greedy choice:

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



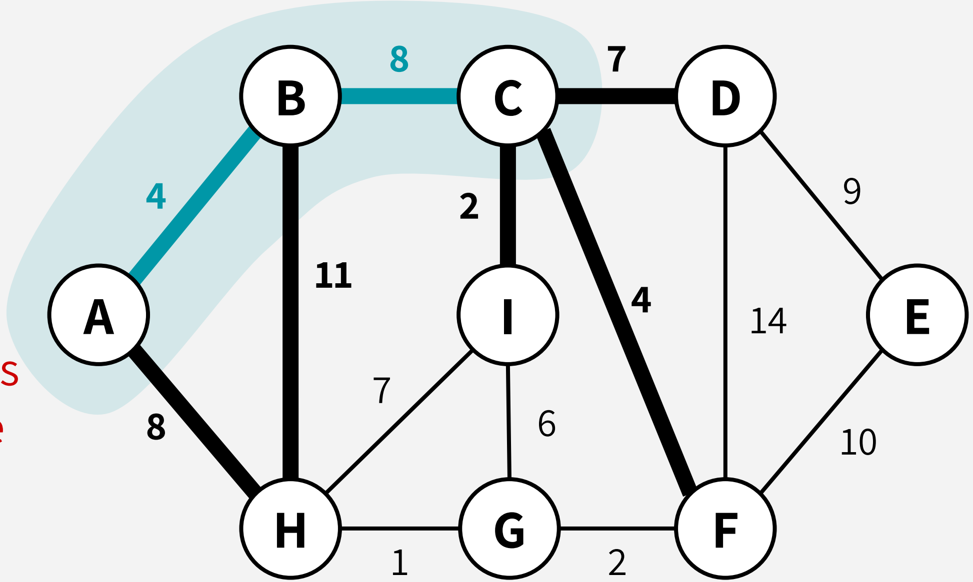
Claim the edge coming out of the "frontier" with the smallest weight

(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

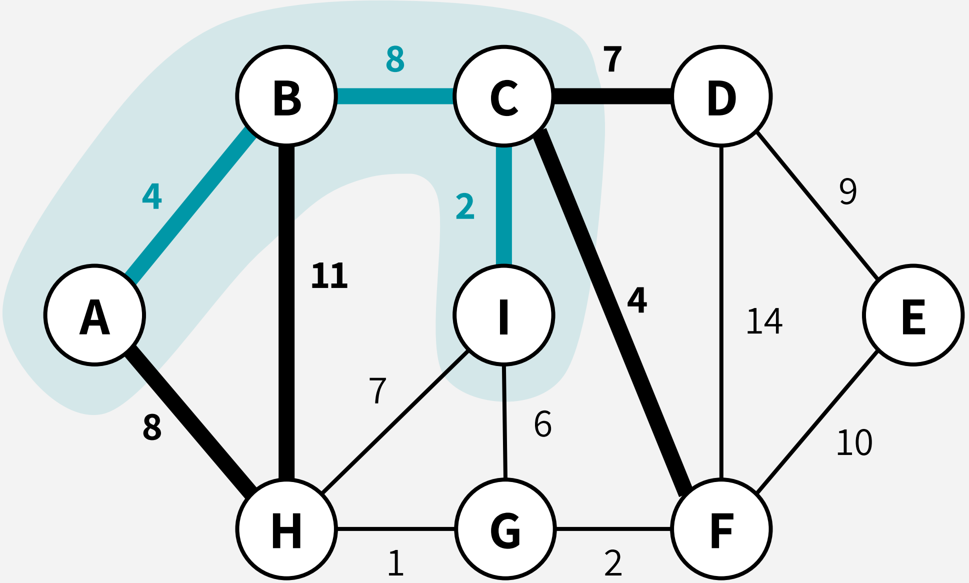


Consider the edges coming out of the "frontier" of our growing tree.

PRIM'S ALGORITHM: THE IDEA

Greedy choice:

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

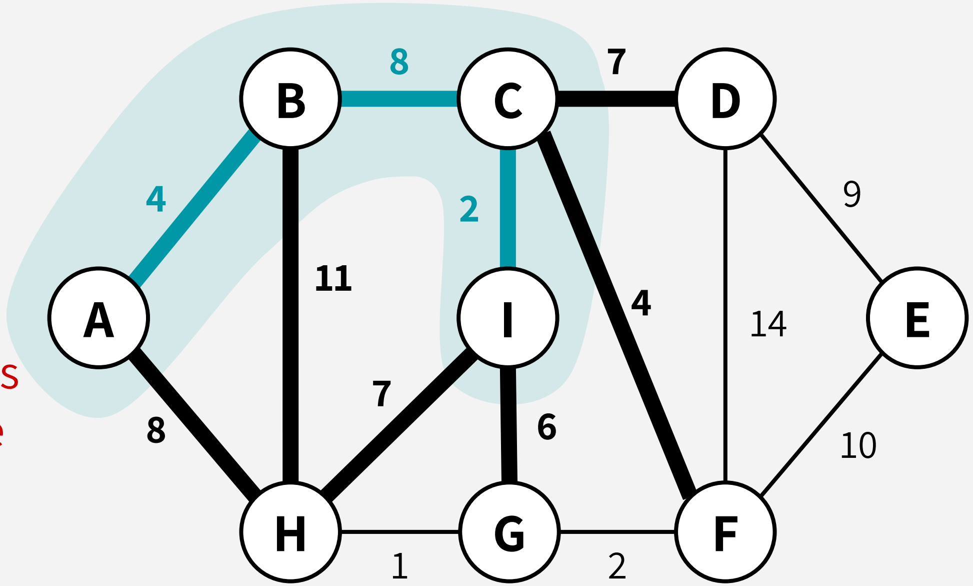


Claim the edge coming out of the "frontier" with the smallest weight

PRIM'S ALGORITHM: THE IDEA

Greedy choice:

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



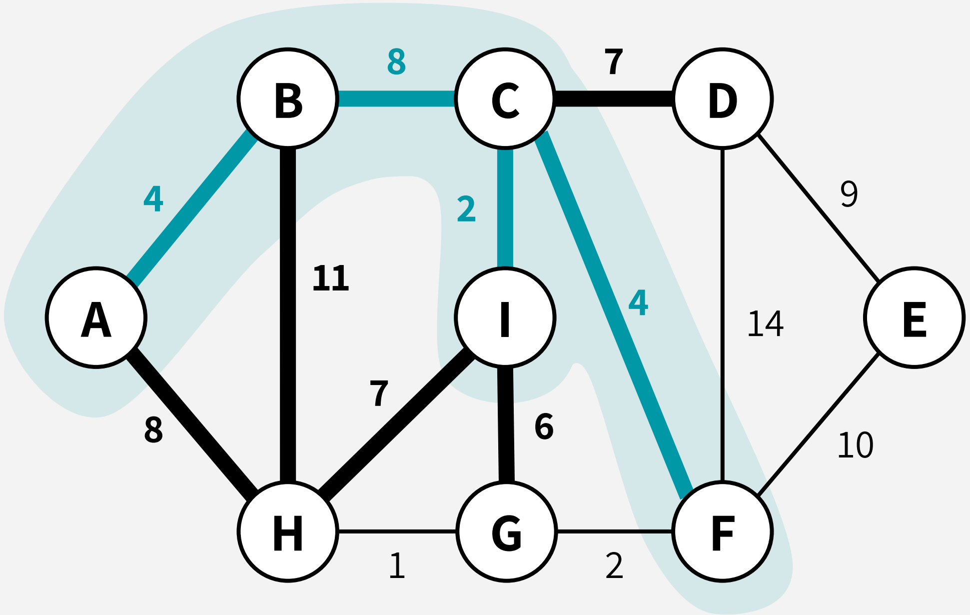
Consider the edges coming out of the "frontier" of our growing tree.

PRIM'S ALGORITHM: THE IDEA

Greedy choice:

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

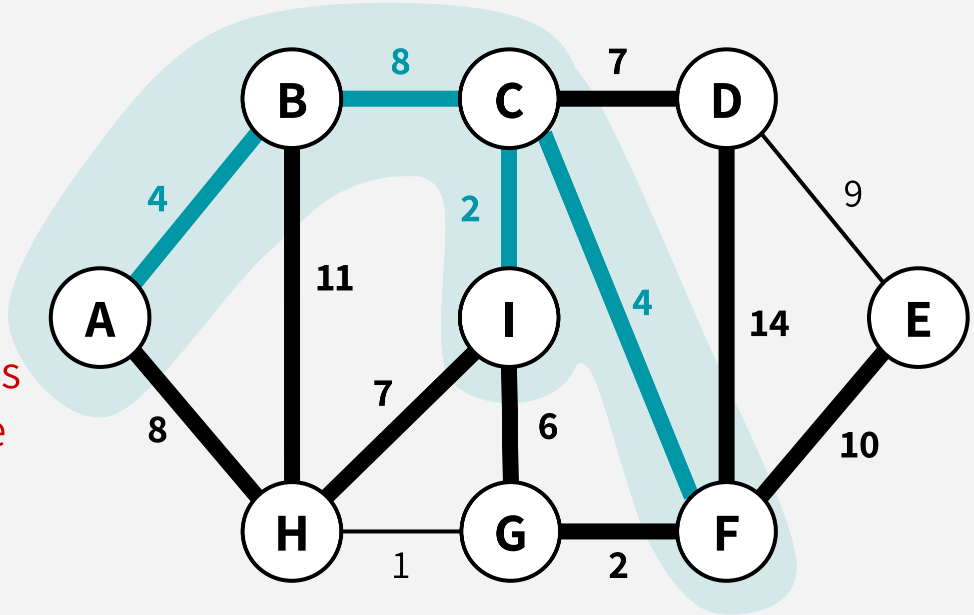
Claim the edge coming out of the "frontier" with the smallest weight



PRIM'S ALGORITHM: THE IDEA

Greedy choice:

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



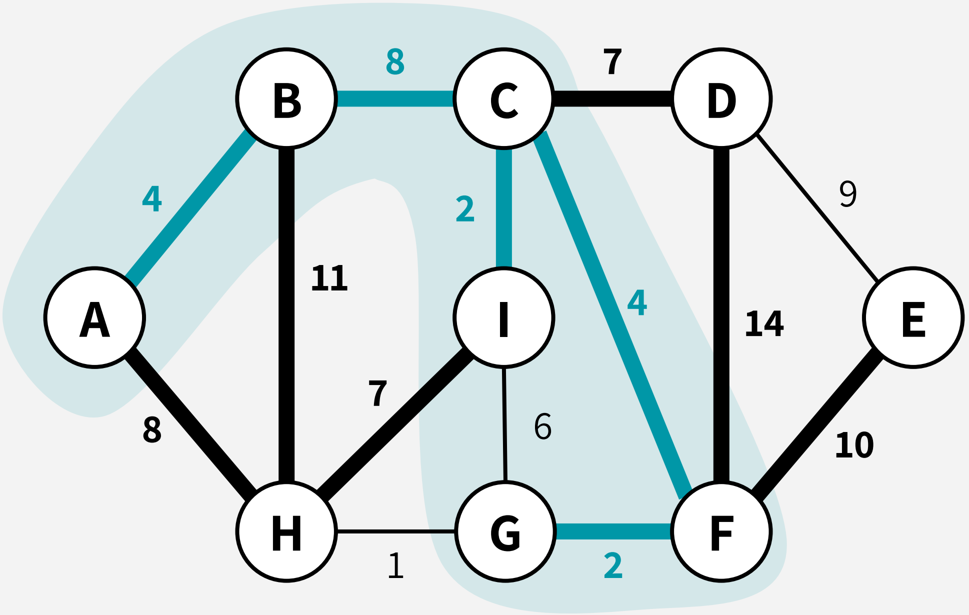
Consider the edges coming out of the "frontier" of our growing tree.

PRIM'S ALGORITHM: THE IDEA

Greedy choice:

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

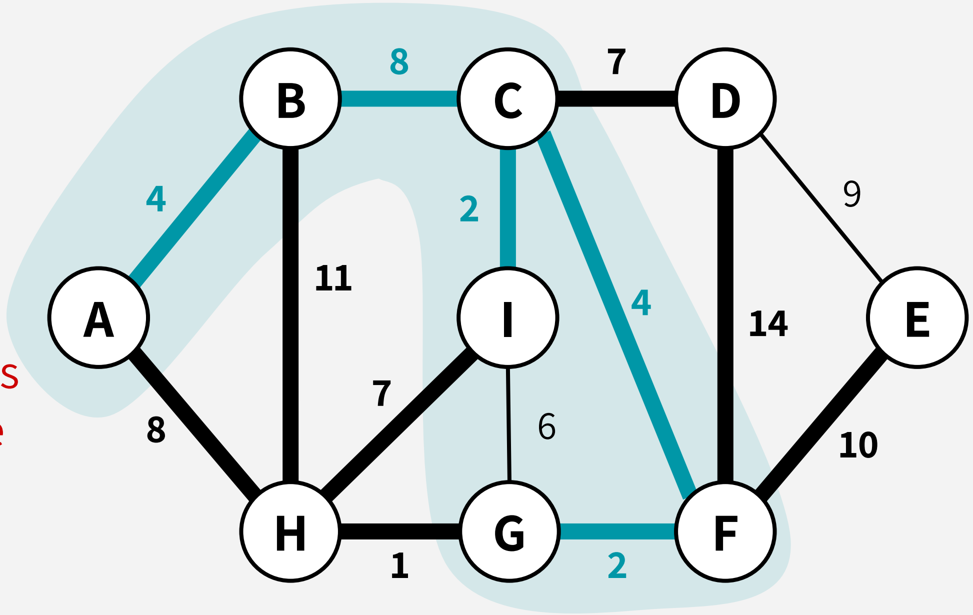
Claim the edge coming out of the "frontier" with the smallest weight



PRIM'S ALGORITHM: THE IDEA

Greedy choice:

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



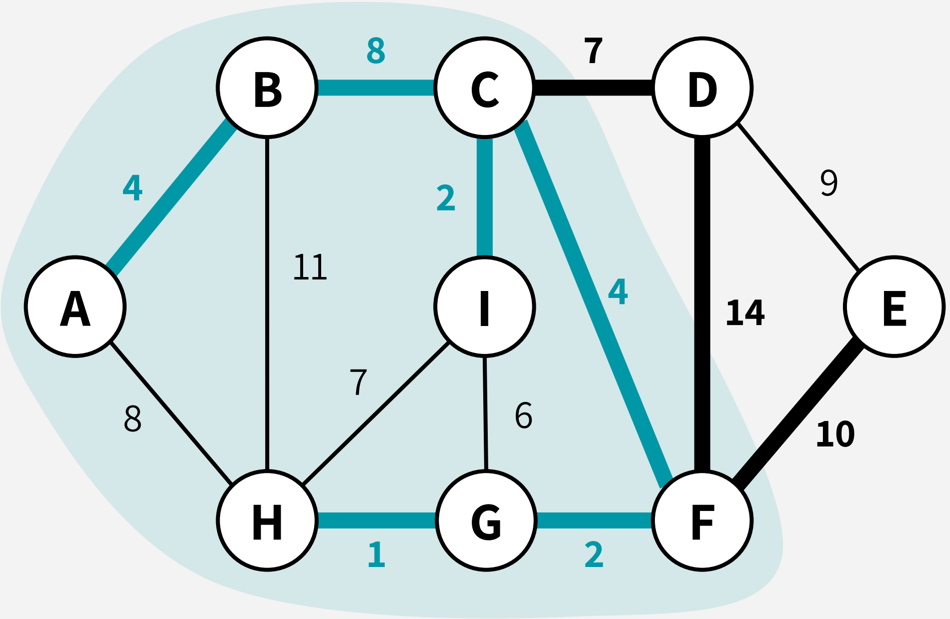
Consider the edges coming out of the "frontier" of our growing tree.

PRIM'S ALGORITHM: THE IDEA

Greedy choice:

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

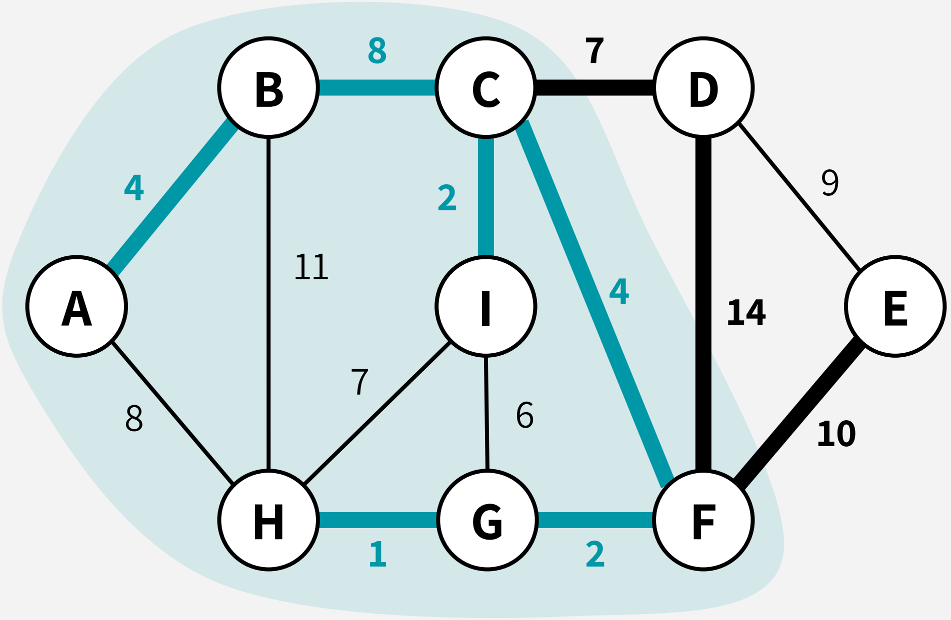
Claim the edge coming out of the "frontier" with the smallest weight



PRIM'S ALGORITHM: THE IDEA

Greedy choice:

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



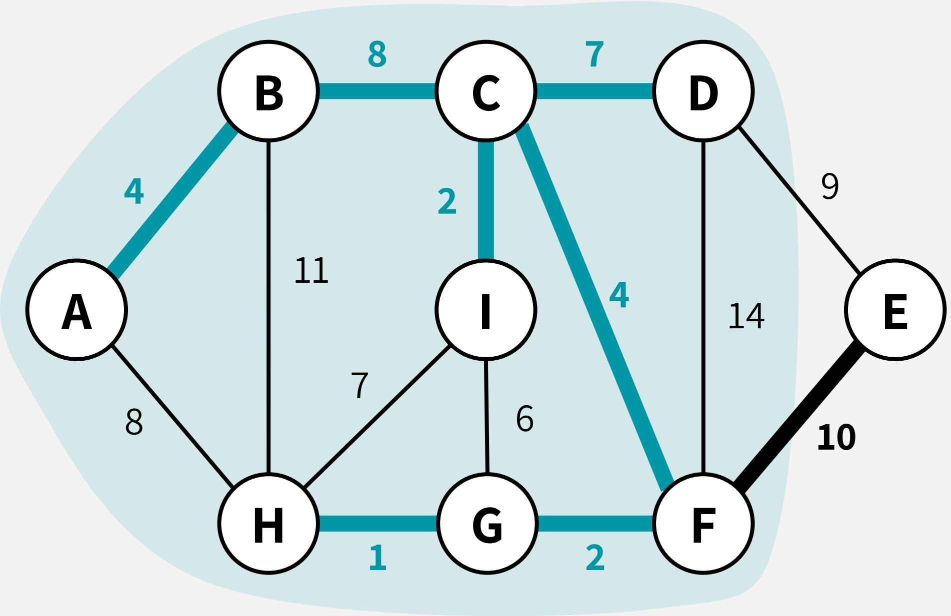
Consider the edges coming out of the "frontier" of our growing tree.

PRIM'S ALGORITHM: THE IDEA

Greedy choice:

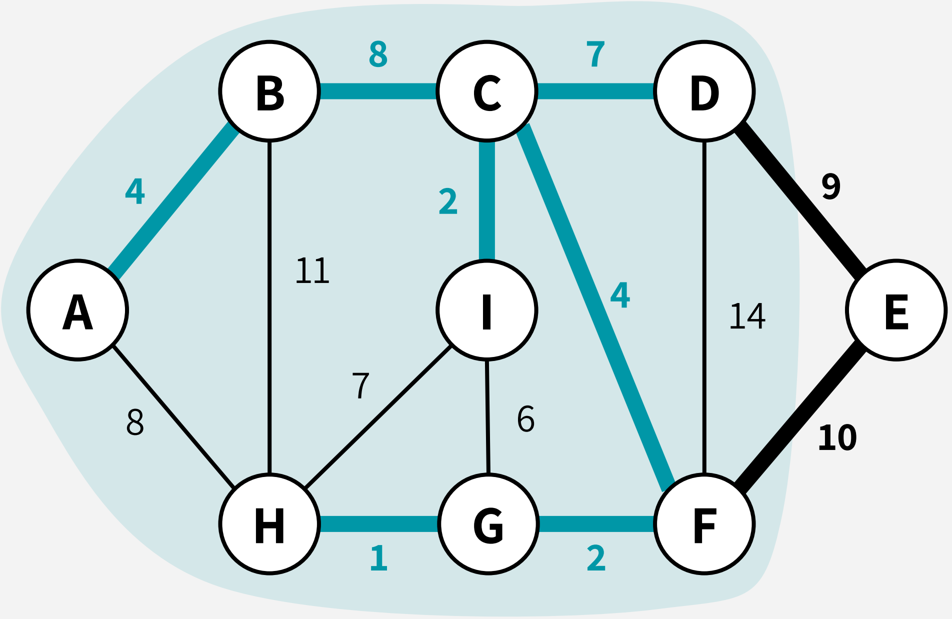
Grow a single tree, & greedily add the shortest edge that could grow our tree

Claim the edge coming out of the "frontier" with the smallest weight



PRIM'S ALGORITHM: THE IDEA

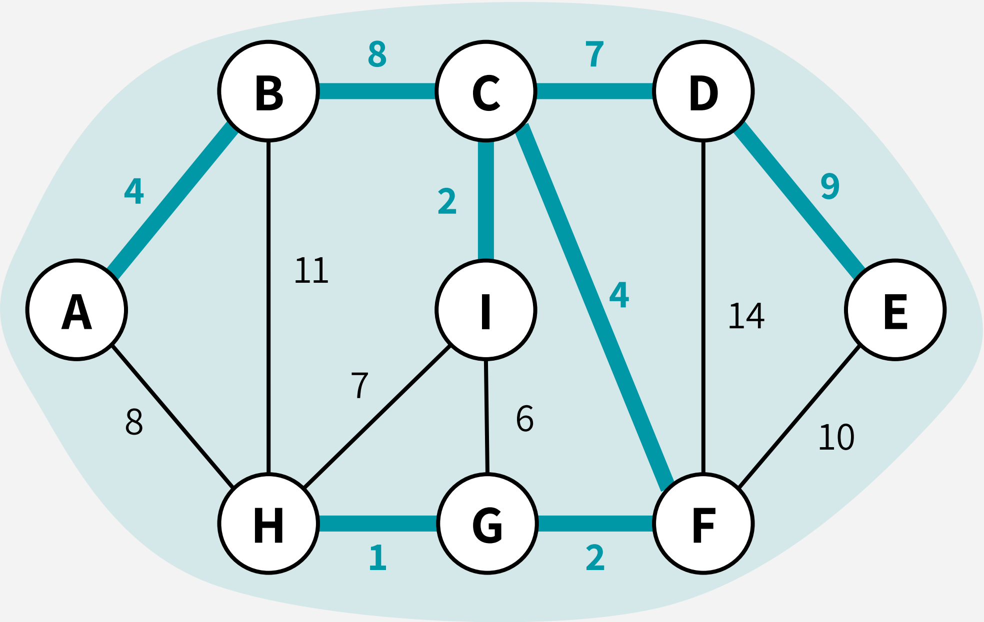
Greedy choice:
Grow a single tree, & greedily add the shortest edge that could grow our tree



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

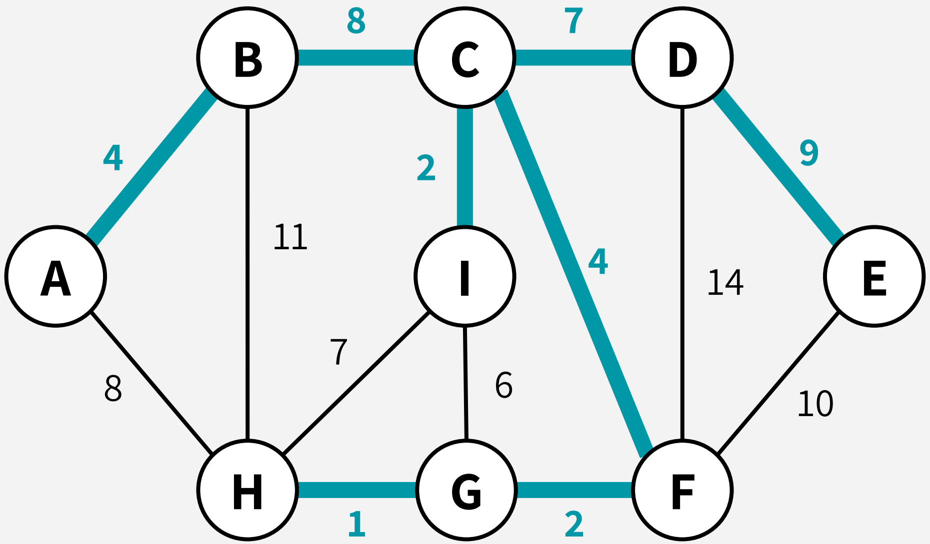


Claim the edge coming out of the "frontier" with the smallest weight

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
```

d the
ch
e O(m)
ion..

(Naive) Runtime: O(nm)

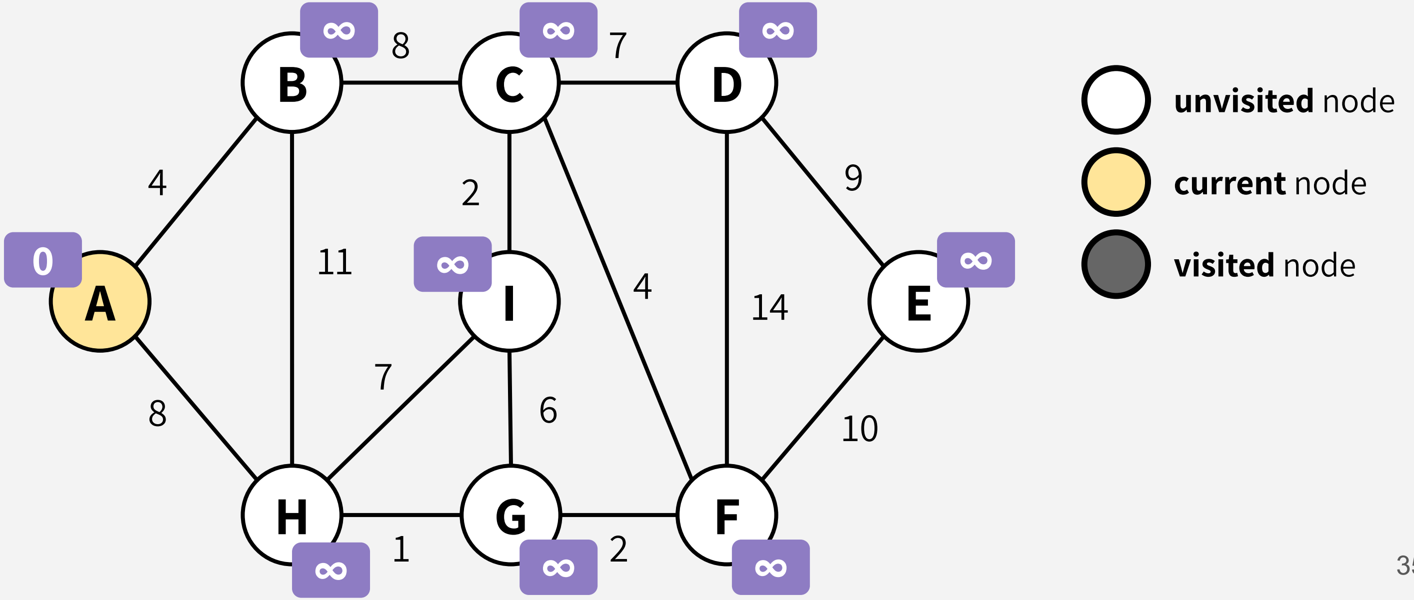
(We'll speed this up by using smart data structures...)

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

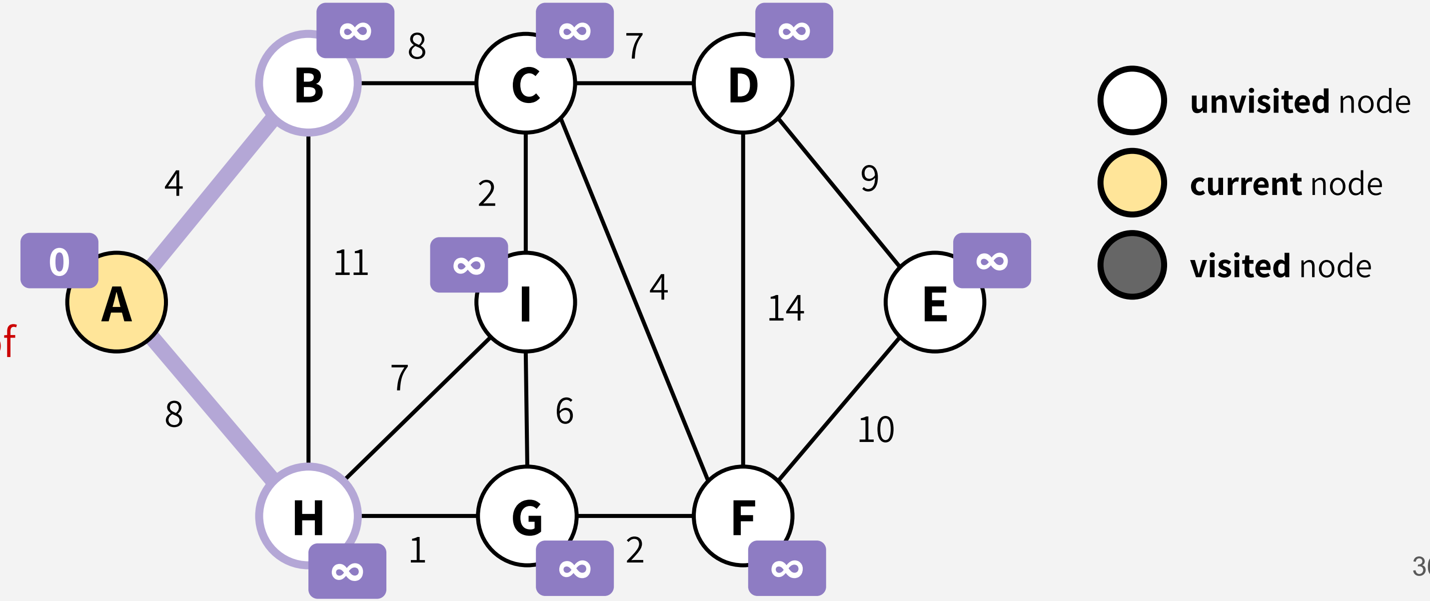


HOW DO WE IMPLEMENT THIS?

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!



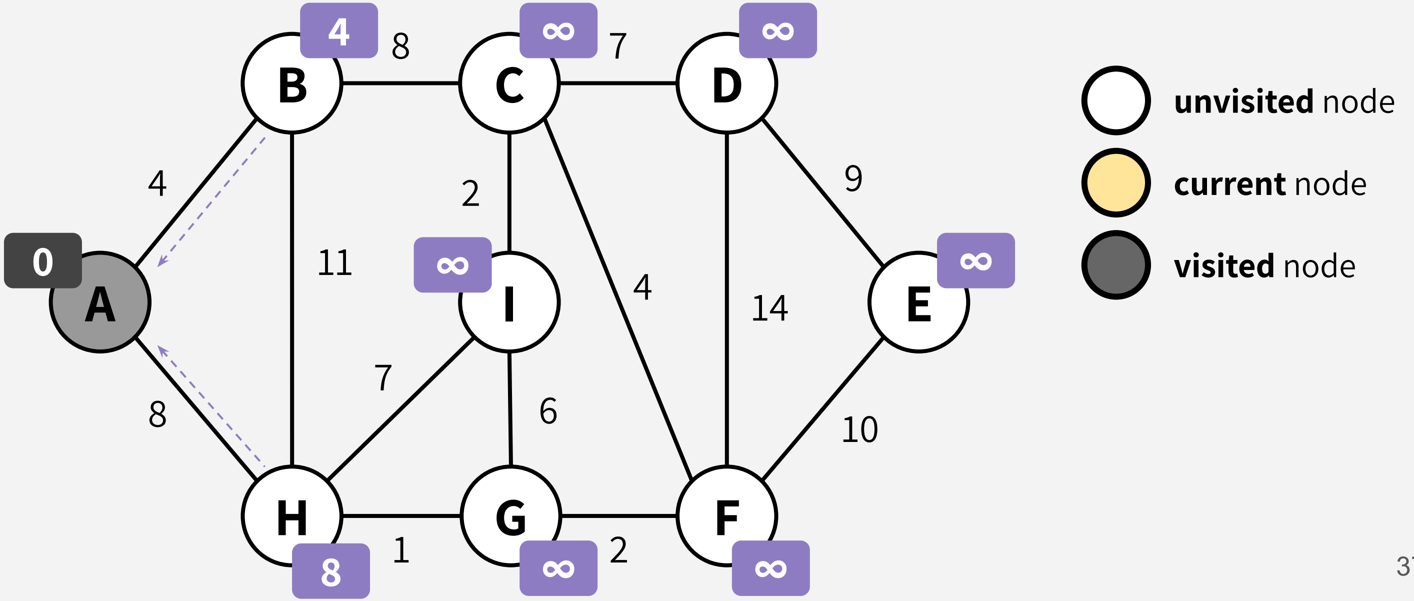
HOW DO WE IMPLEMENT THIS?

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)



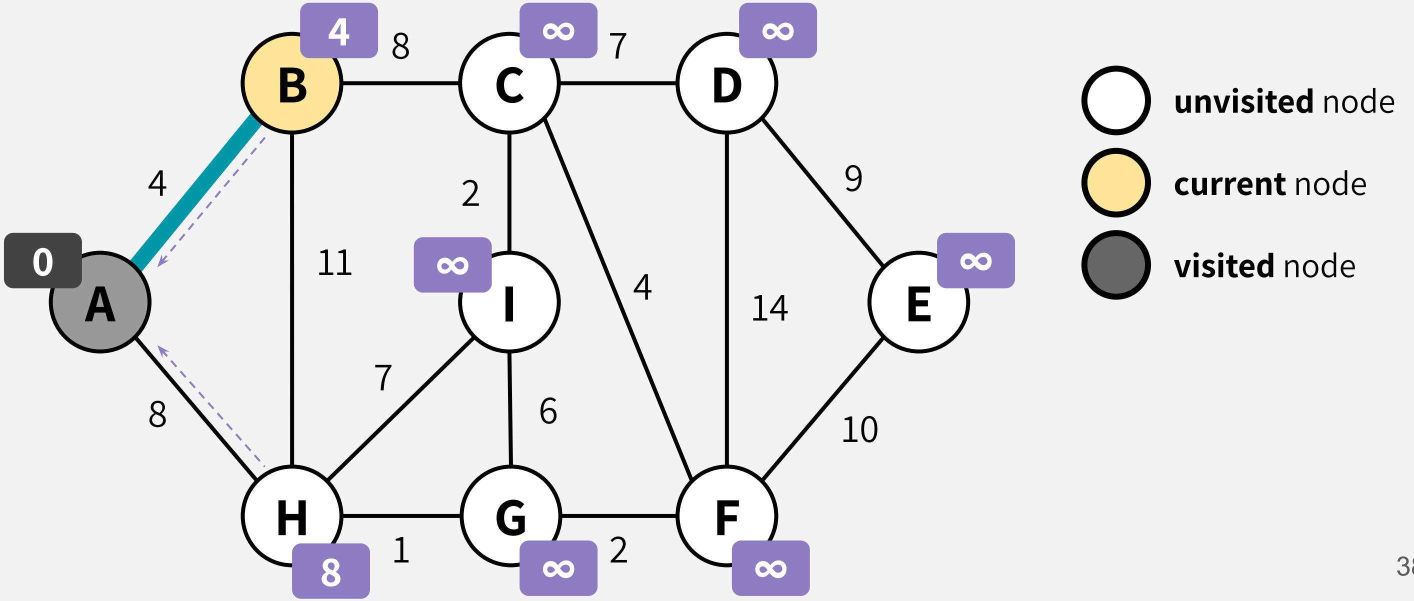
HOW DO WE IMPLEMENT THIS?

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)

B is the closest node to the growing tree.

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

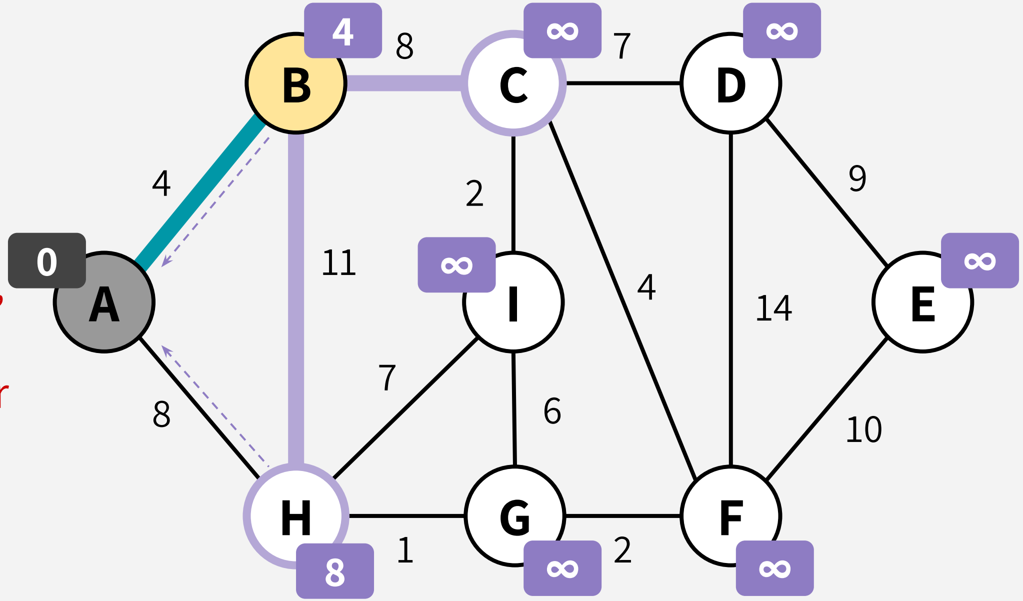


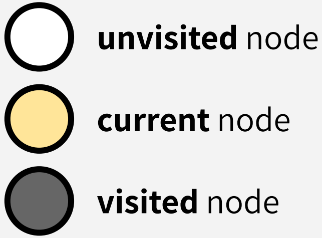
HOW DO WE IMPLEMENT THIS?

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





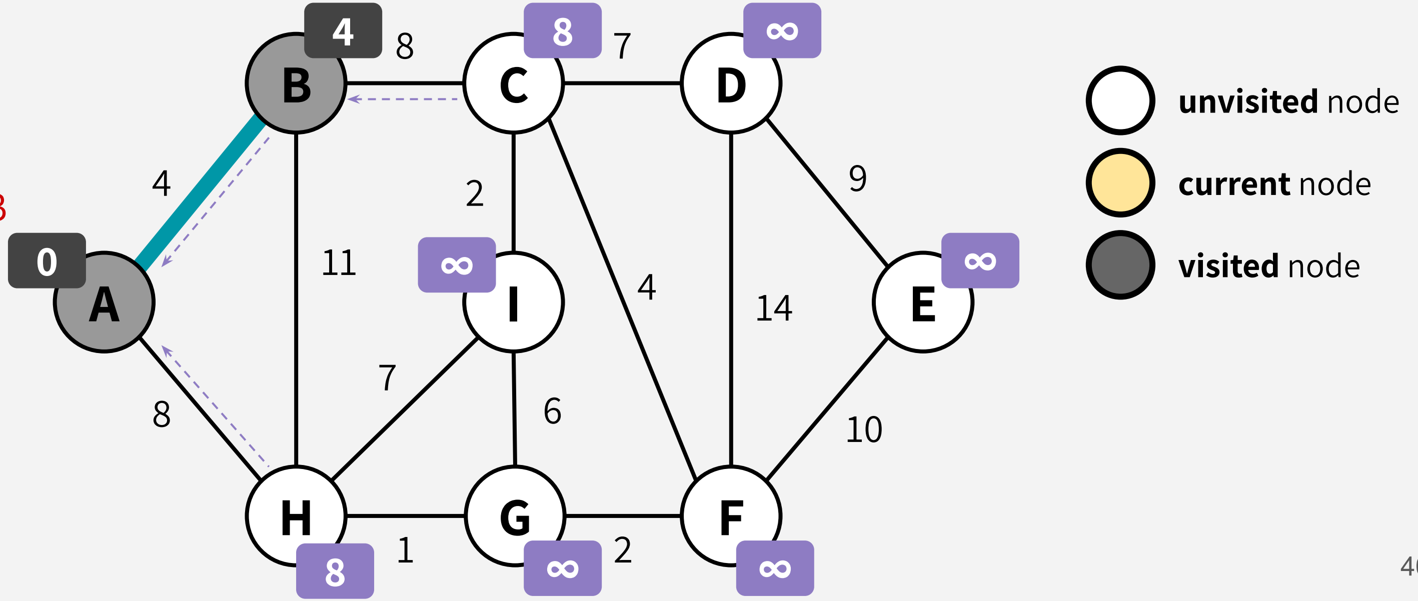
HOW DO WE IMPLEMENT THIS?

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)



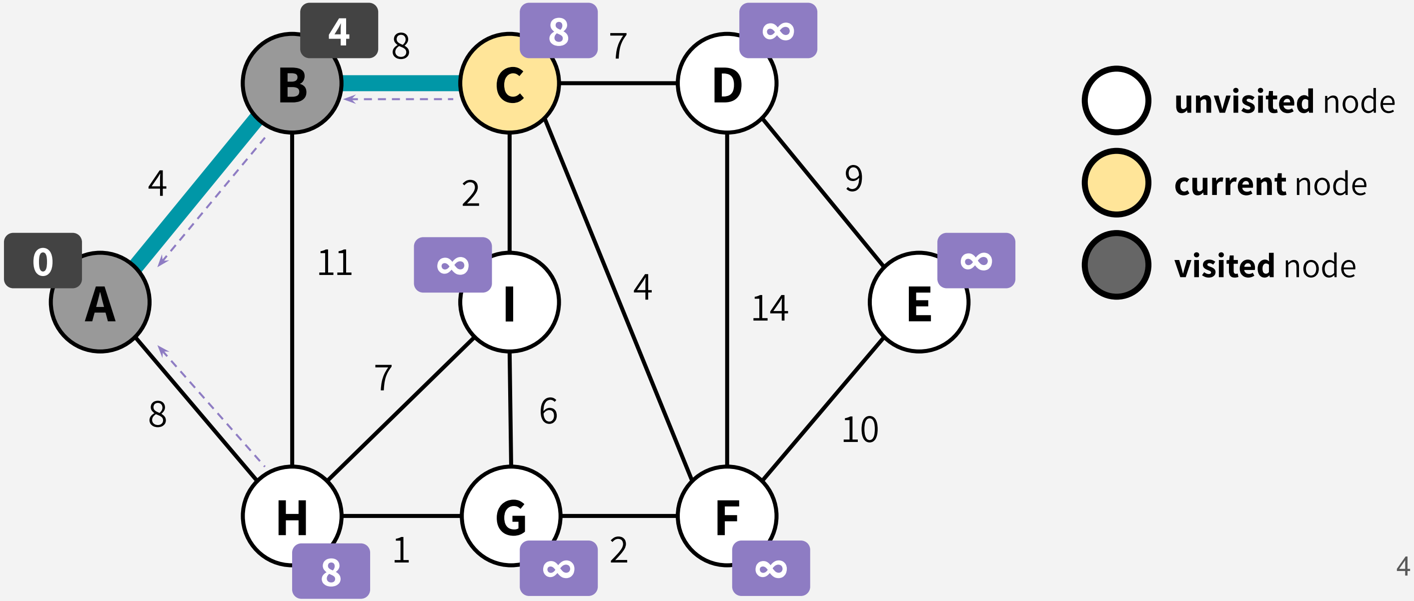
HOW DO WE IMPLEMENT THIS?

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.

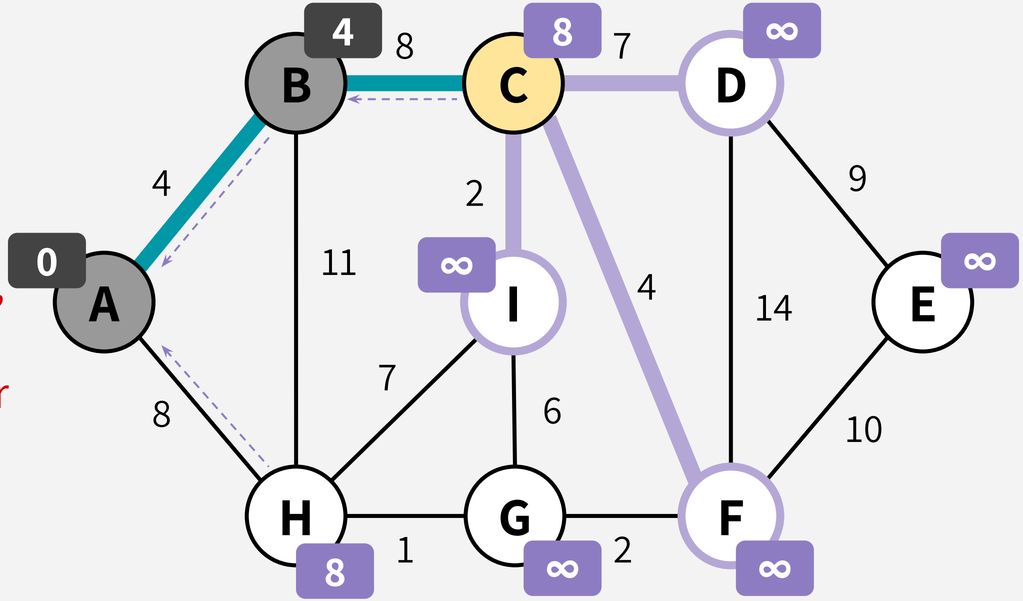


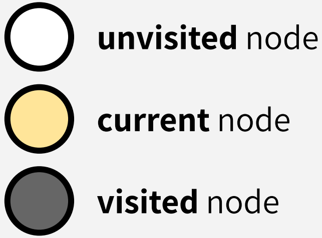
HOW DO WE IMPLEMENT THIS?

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!





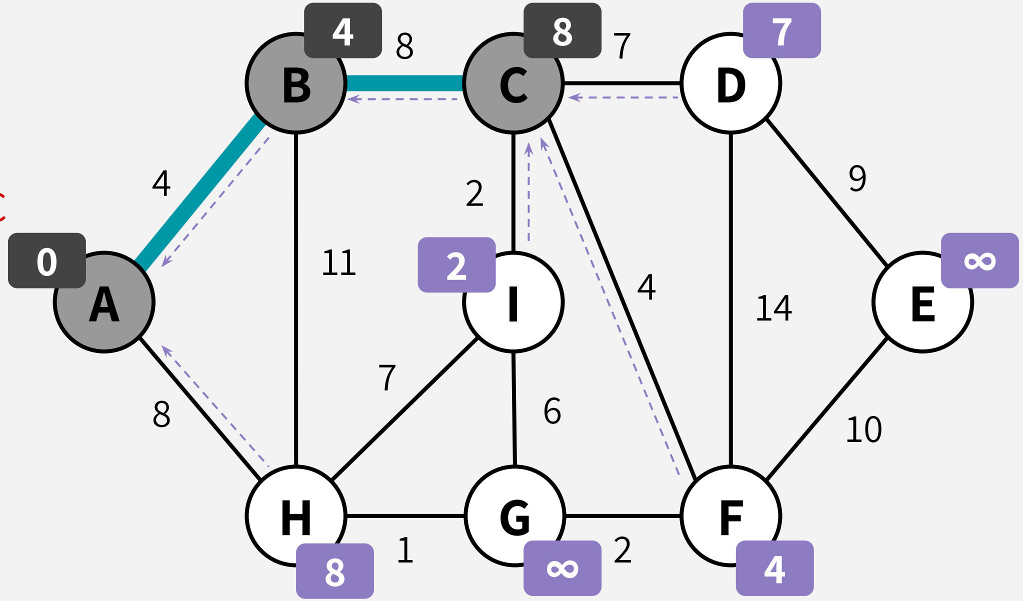
HOW DO WE IMPLEMENT THIS?

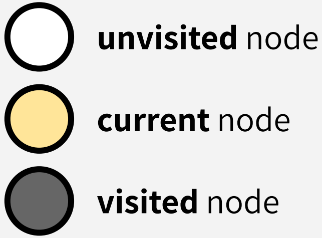
Each vertex that's not yet reached by the growing tree keeps track of:

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)





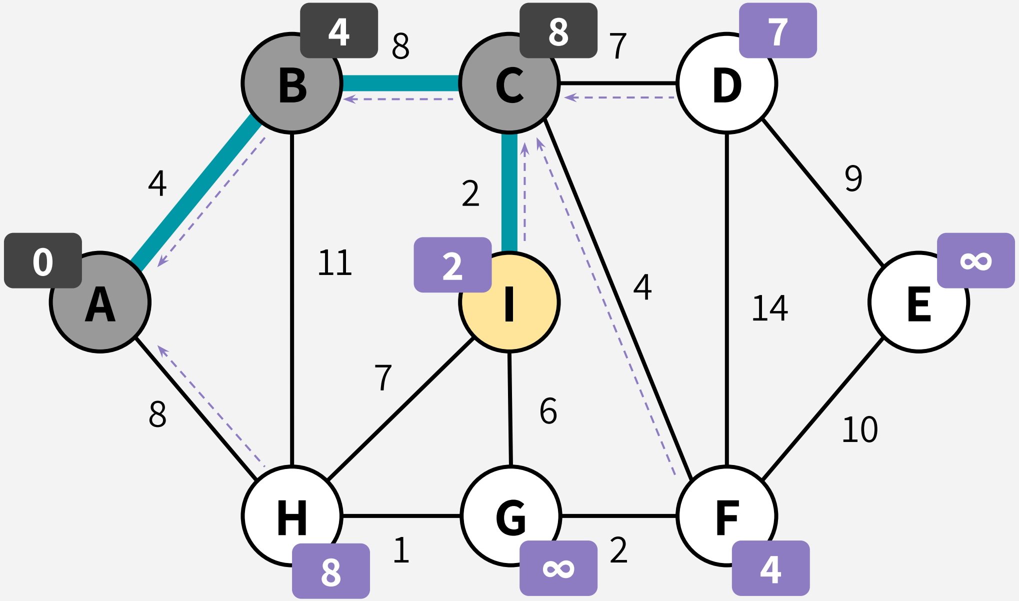
HOW DO WE IMPLEMENT THIS?

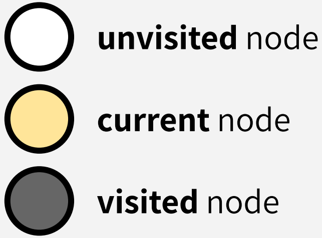
Each vertex that's not yet reached by the growing tree keeps track of:

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.



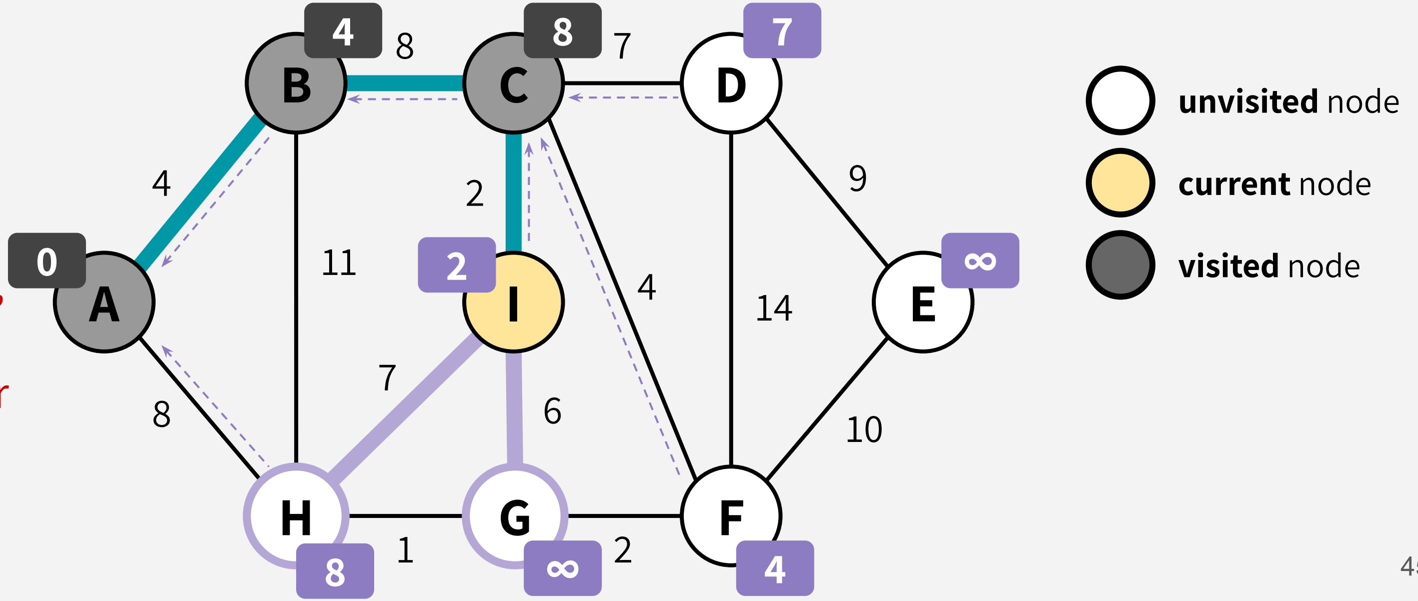


HOW DO WE IMPLEMENT THIS?

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!



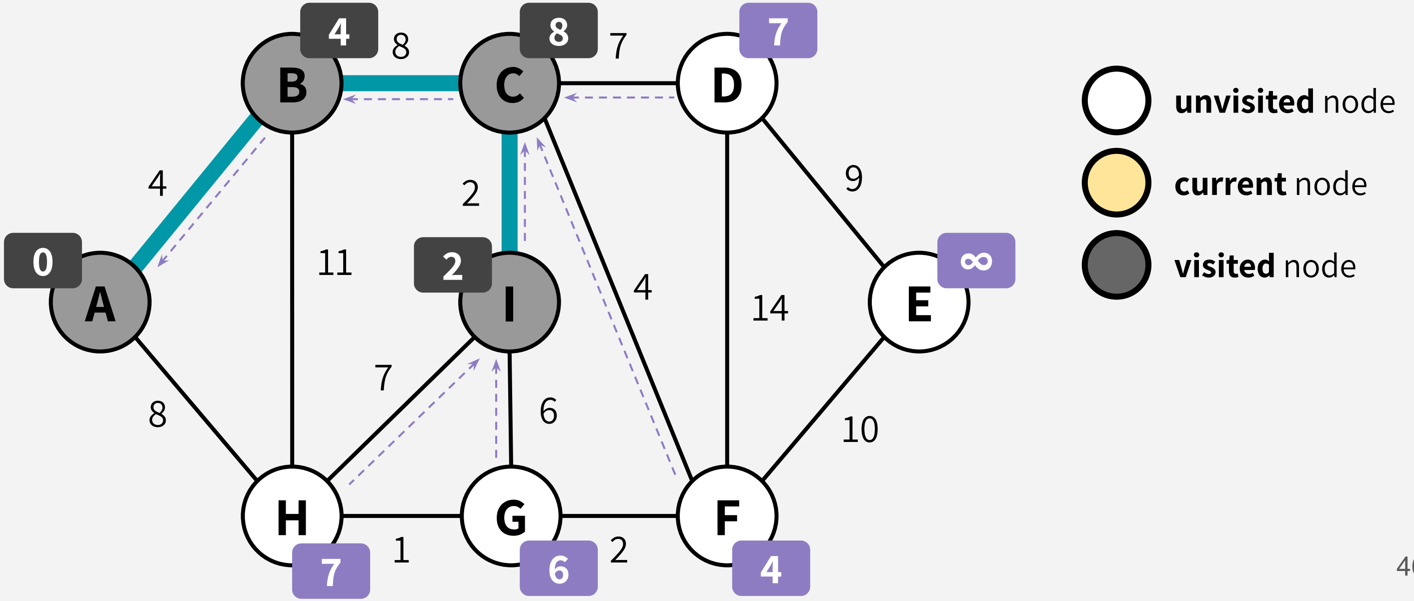
HOW DO WE IMPLEMENT THIS?

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] = ∞ and k[v] = NULL
    for each neighbor v of s: d[v] = w(s,v) and k[v] = s
    while 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 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! Remember, Dijkstra's runtime depended 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] = ∞ and k[v] = NULL
    for each neighbor v of s: d[v] = w(s,v) and k[v] = s
    while 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 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! Remember, Dijkstra's runtime depended on the data structure used for a priority queue.)

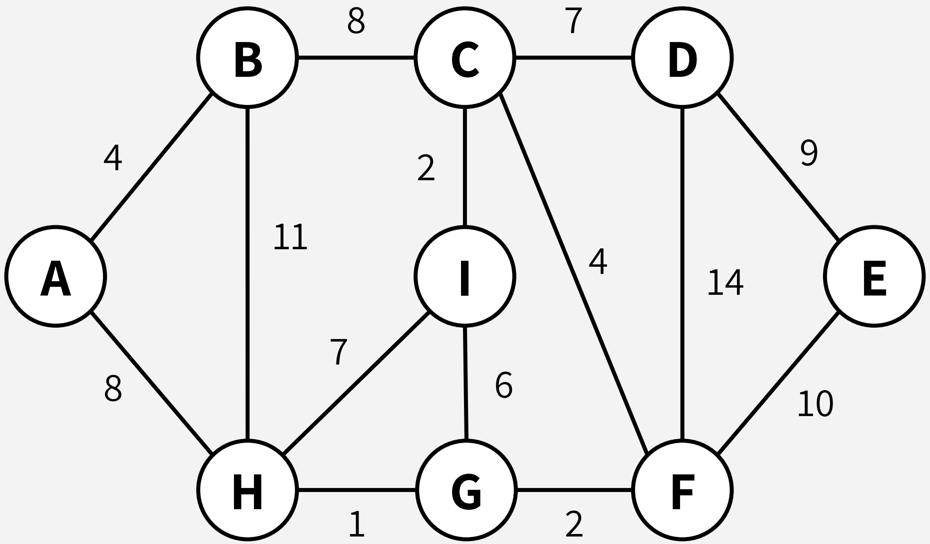
KRUSKAL'S ALGORITHM

Greedily add the cheapest edge!

KRUSKAL'S ALGORITHM: THE IDEA

Greedy choice:

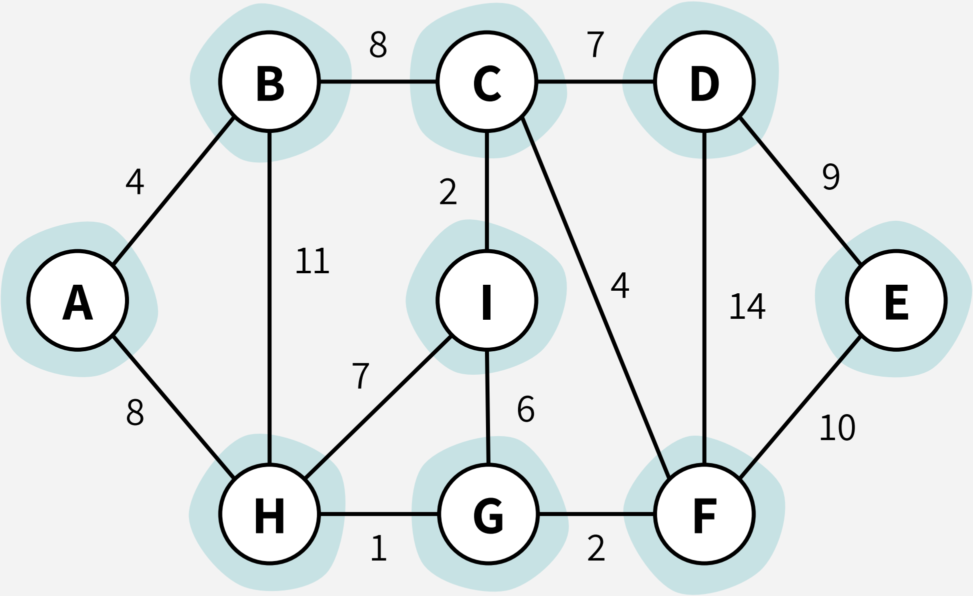
Maintain a forest of trees, & greedily add the cheapest edge to combine trees



KRUSKAL'S ALGORITHM: THE IDEA

Greedy choice:

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

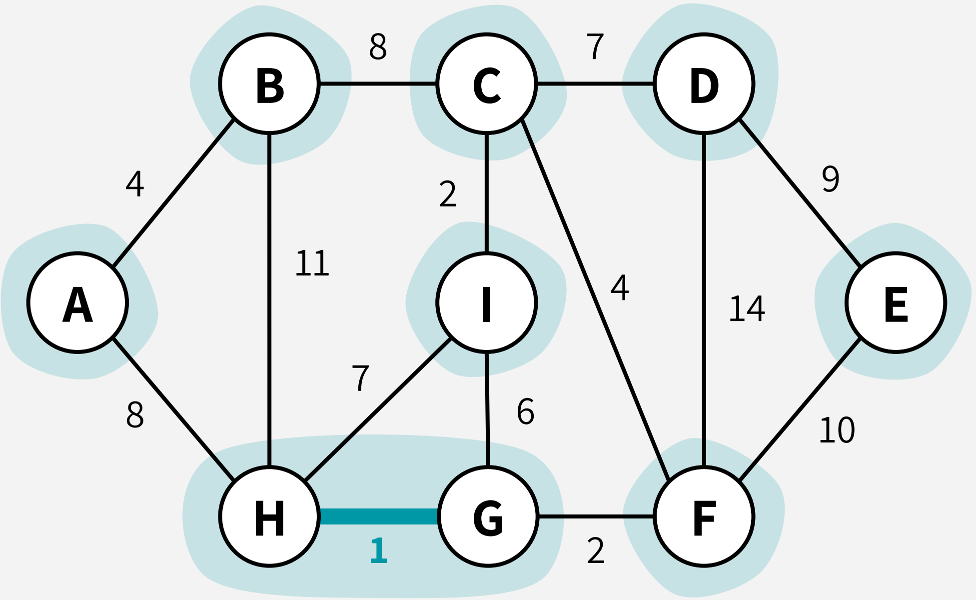


Every node on its own starts as an individual tree in this forest

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)



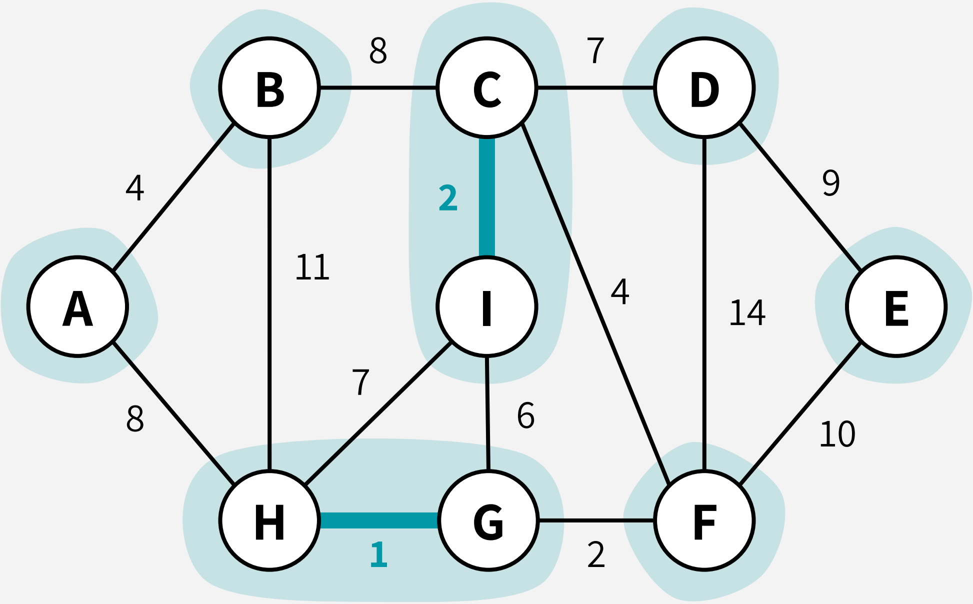
KRUSKAL'S ALGORITHM: THE IDEA

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)

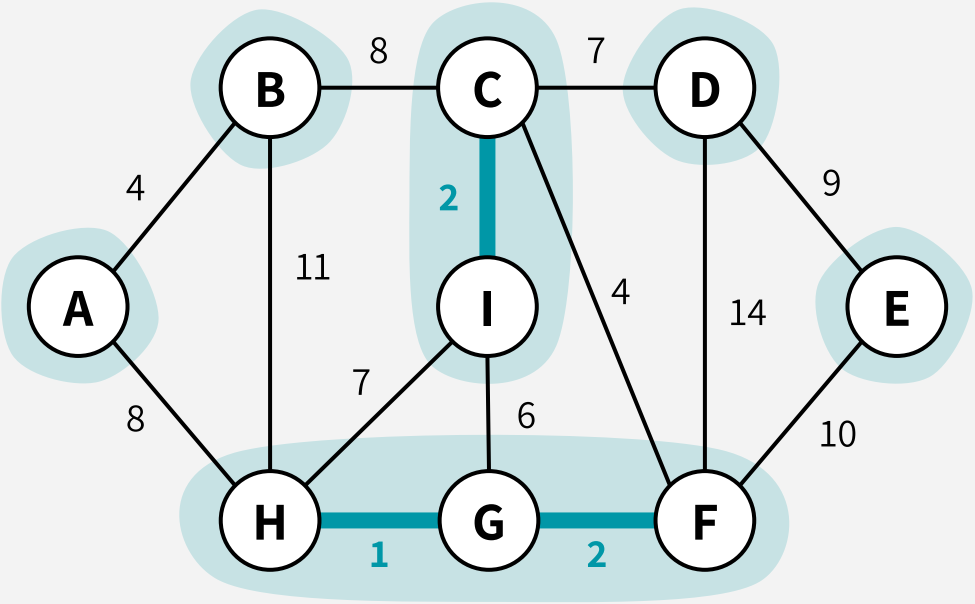


If there's a tie, choose one of the edges

KRUSKAL'S ALGORITHM: THE IDEA

Greedy choice:

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



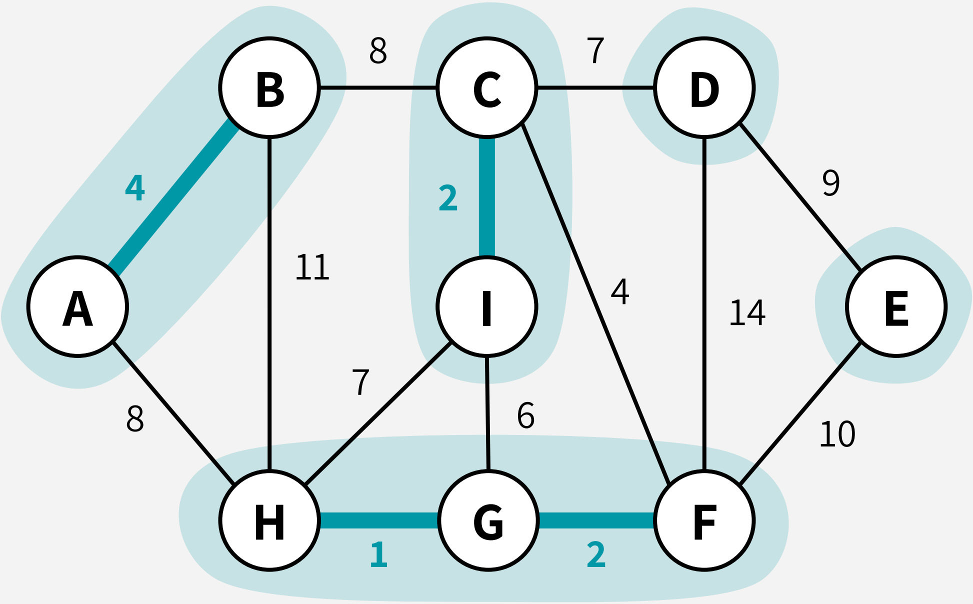
Choose the cheapest edge that would combine two trees

(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

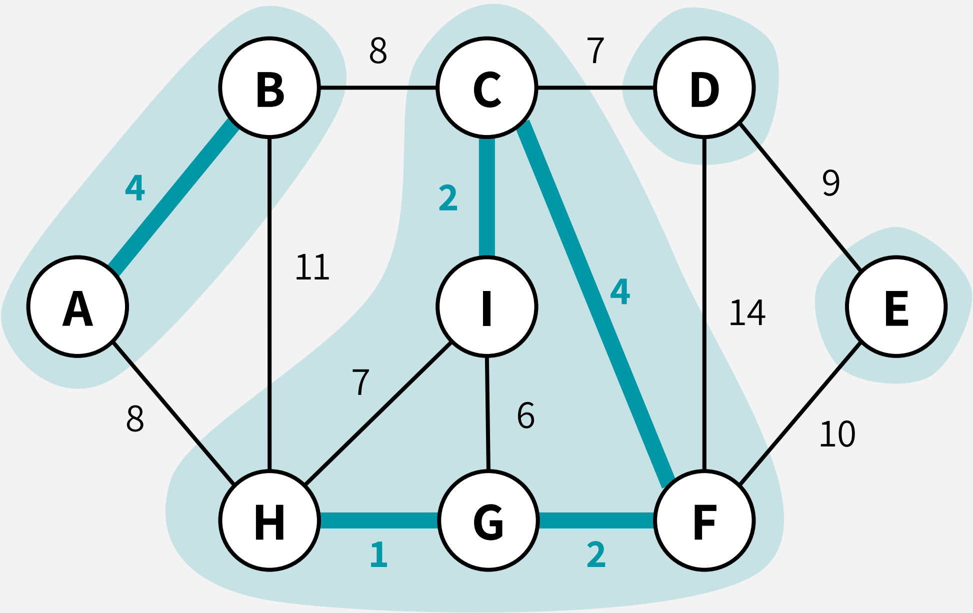
(i.e. that won't cause a cycle)

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)

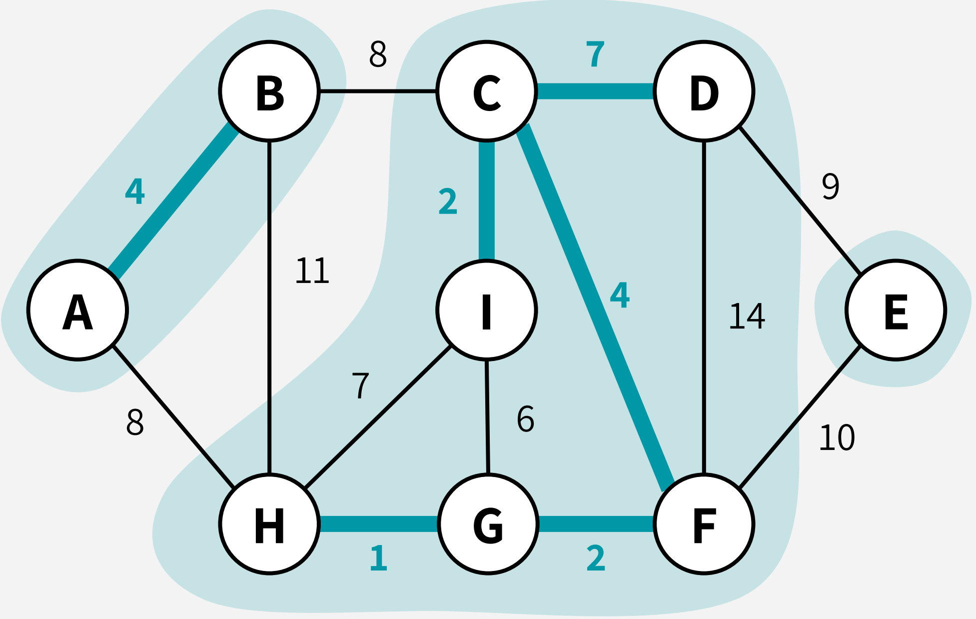
KRUSKAL'S ALGORITHM: THE IDEA

Greedy choice:

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

Choose the cheapest edge that would combine two trees

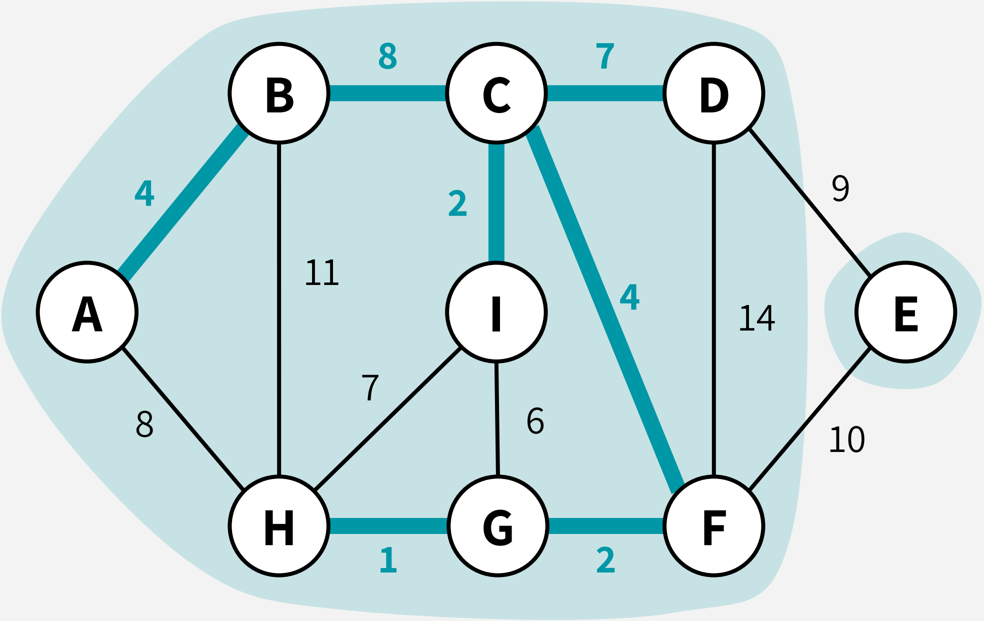
(i.e. that won't cause a cycle)



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)

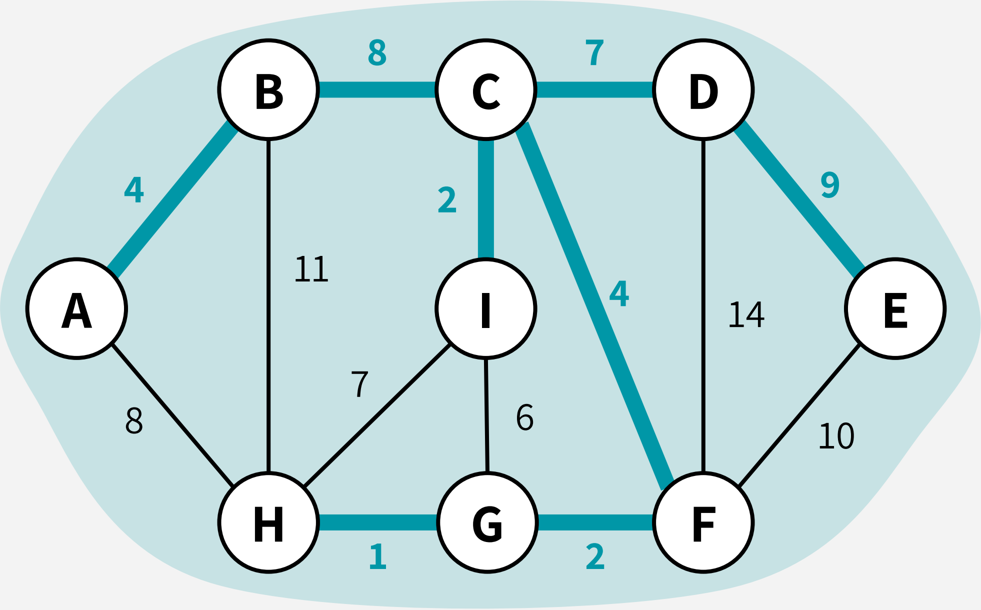
KRUSKAL'S ALGORITHM: THE IDEA

Greedy choice:

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

Choose the cheapest edge that would combine two trees

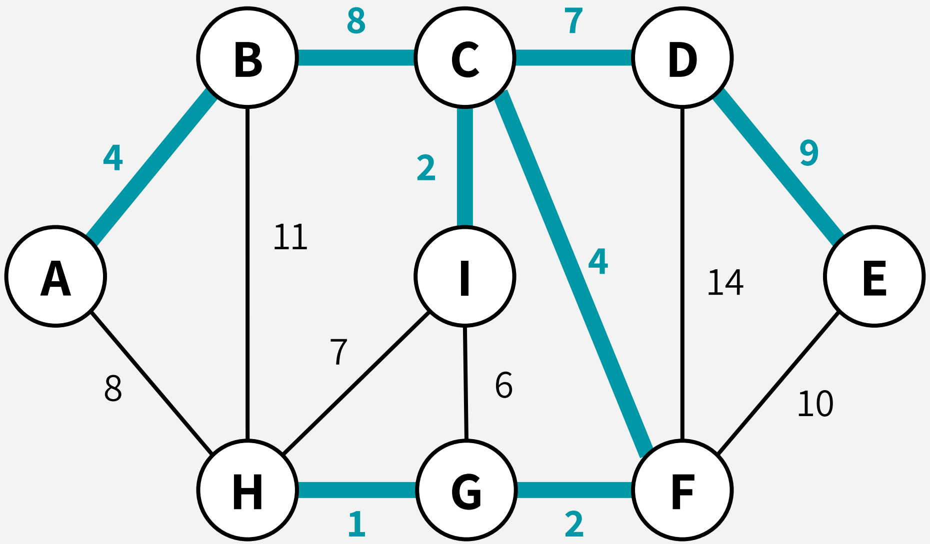
(i.e. that won't cause a cycle)



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
```

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 ~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)

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

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