CS 2009
Design and Analysis of Algorithms

Waheed Ahmed

Farrukh Salim Shaikh

THE GREEDY PARADIGM

Commit to choices one-at-a-time,

never look back,

and hope for the best.

Greedy doesn't always work.

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

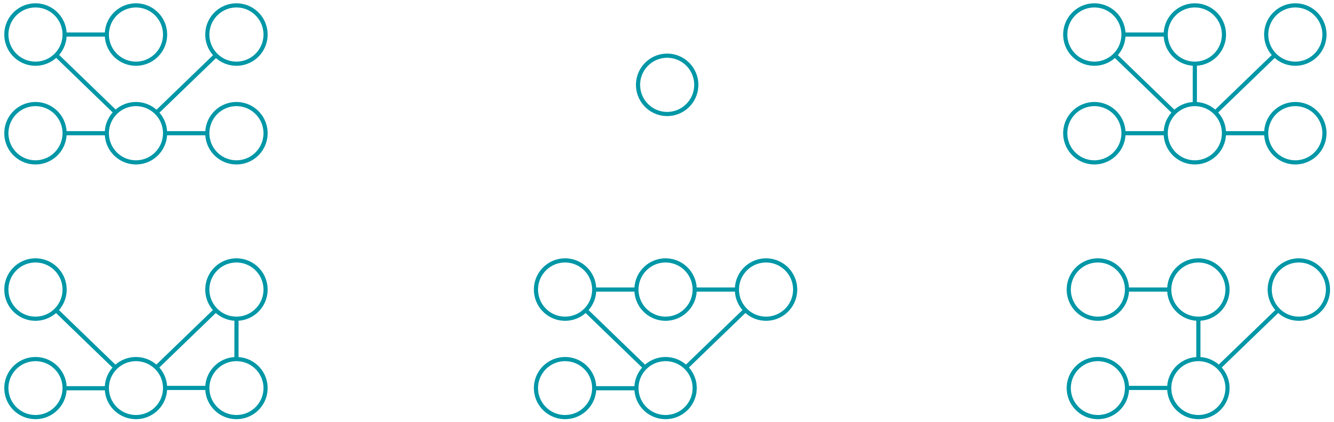
What are minimum spanning trees (MSTs)?

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?

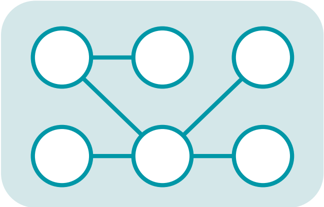


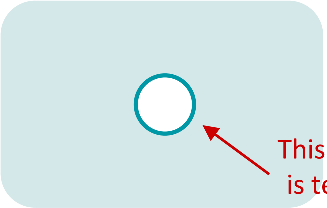
TREES IN GRAPHS

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?



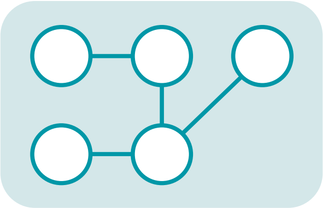


This single node is technically a valid tree!

Contains cycle

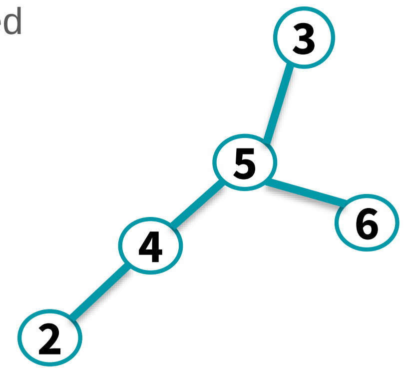
Contains cycle

Contains cycle



TREES IN UNIDIRECTED GRAPHS?

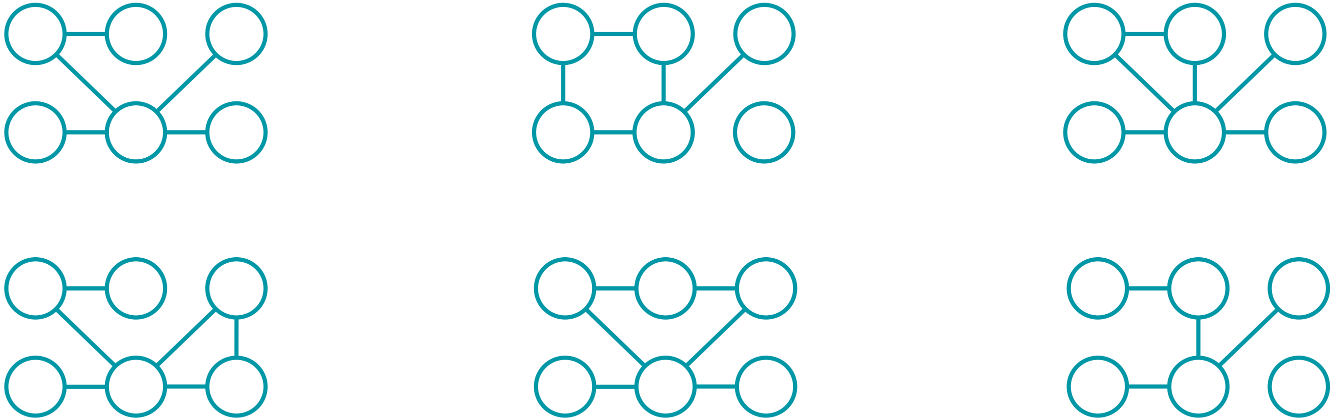
- However, in undirected graphs, there is another definition of trees
- Tree
    - A undirected graph (V, E), where E is the set of undirected edges
    - All vertices are connected
    - $|E|=|V|-1$



SPANNING TREES

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

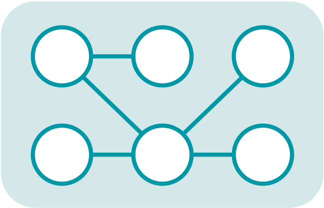
Which of these are spanning trees?



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

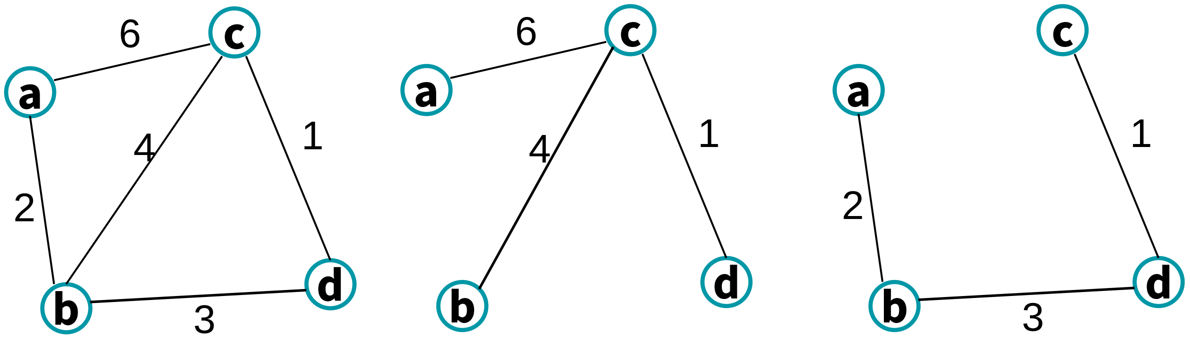
Not a tree

Not a tree

Doesn’t connect all vertices

Examples of MST

Example:



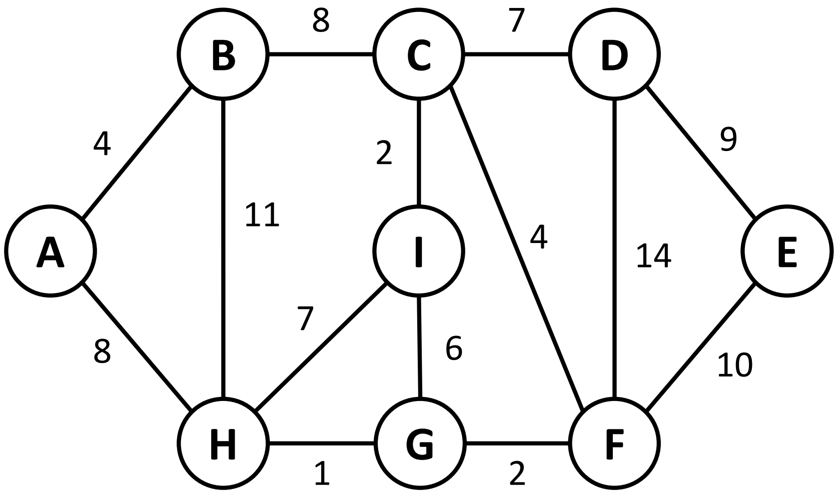
MINIMUM SPANNING TREES (MSTs)

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)



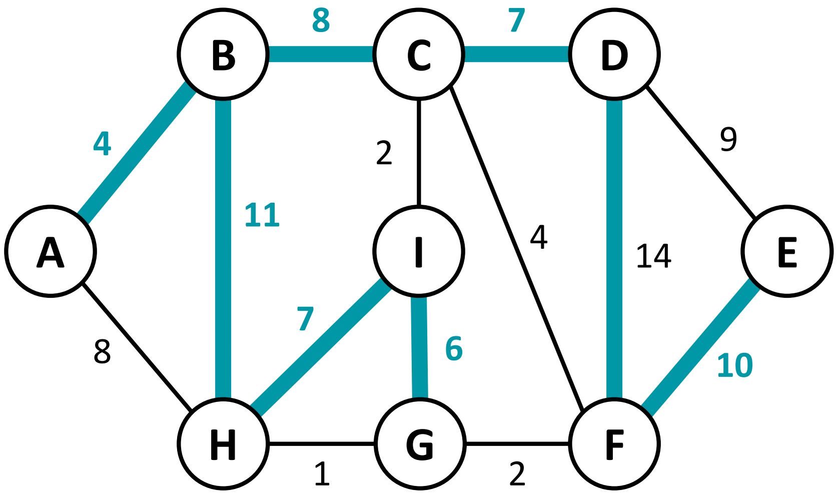
MINIMUM SPANNING TREES (MSTs)

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**.

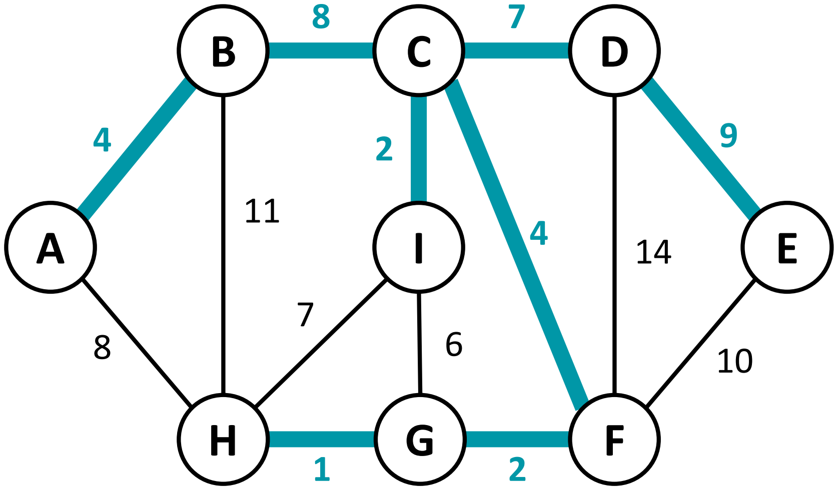
MINIMUM SPANNING TREES (MSTs)

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)



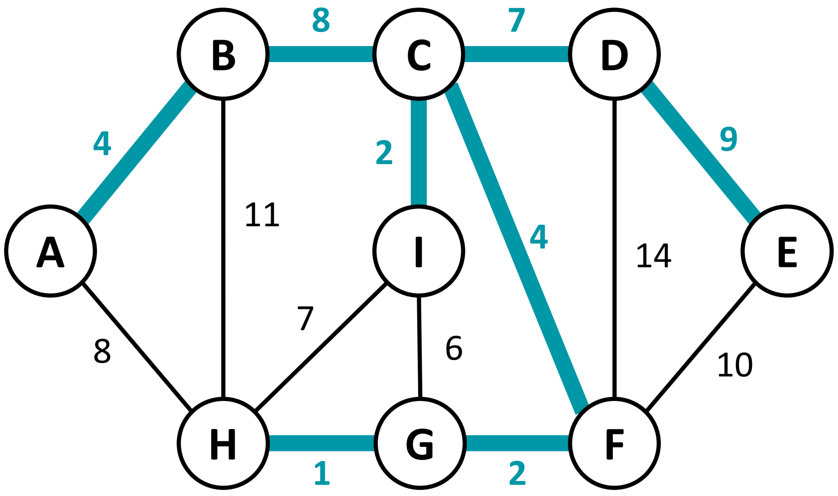
This spanning tree has a cost of **37**.

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.

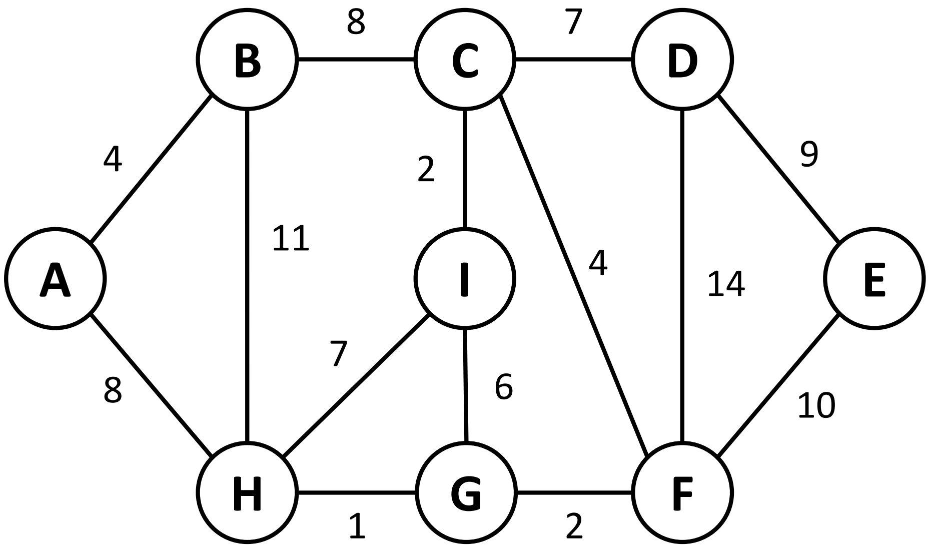
PRIM'S ALGORITHM

Greedily add the closest vertex!

PRIM'S ALGORITHM: THE IDEA

Greedy choice:

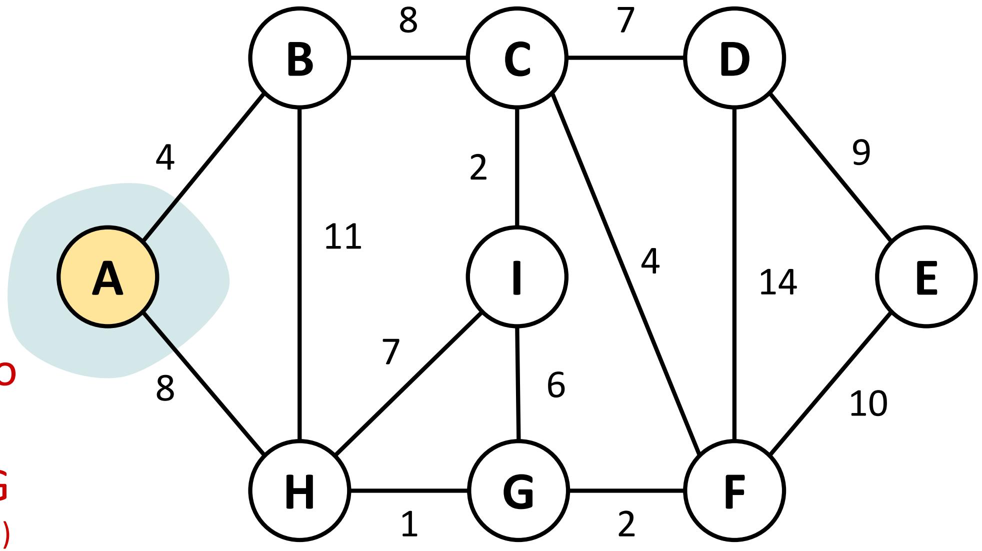
Grow a single tree, & greedily add the shortest edge that could grow our tree



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

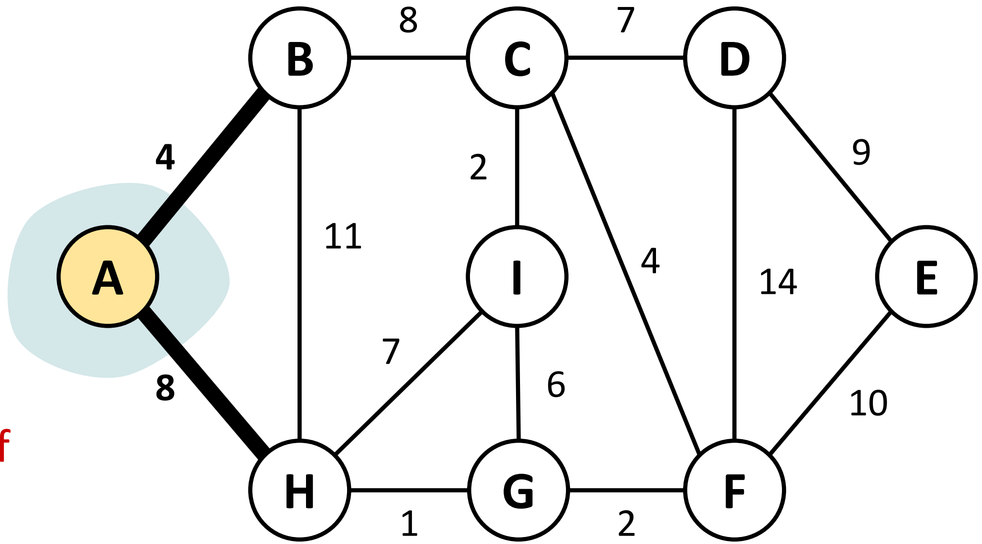
(doesn't matter which node)

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.

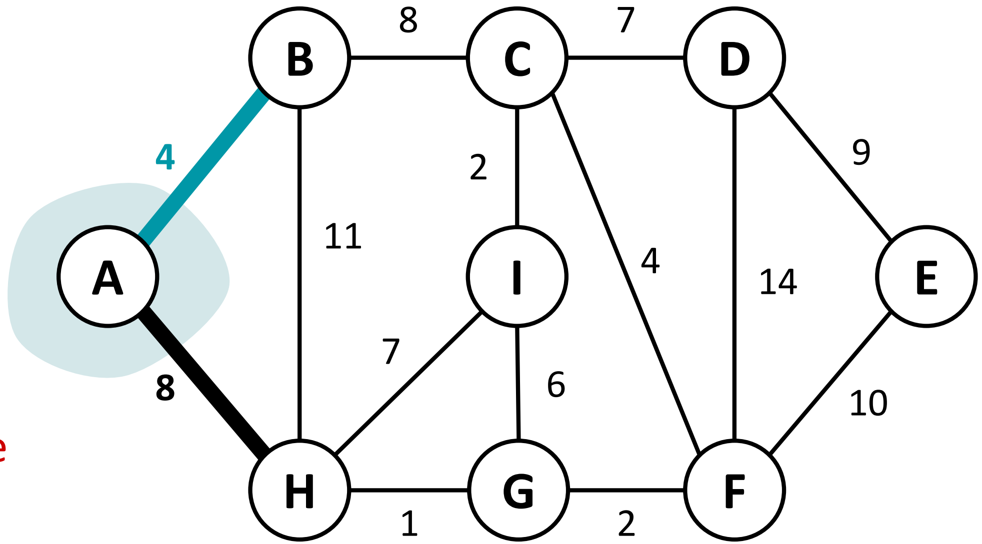


PRIM'S ALGORITHM: THE IDEA

Greedy choice:

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

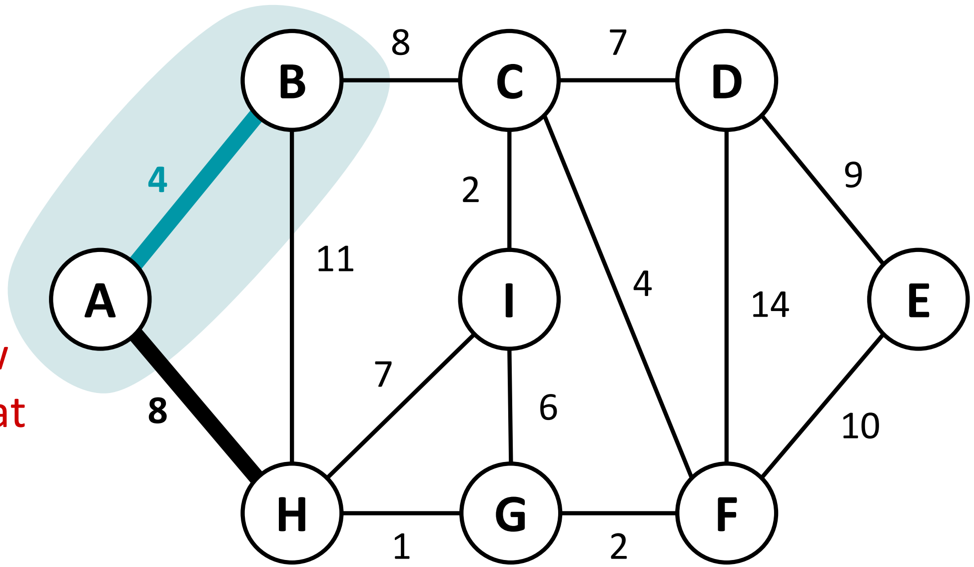
Claim the edge coming out of the "frontier" with the smallest weight



PRIM'S ALGORITHM: THE IDEA

Greedy choice:

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



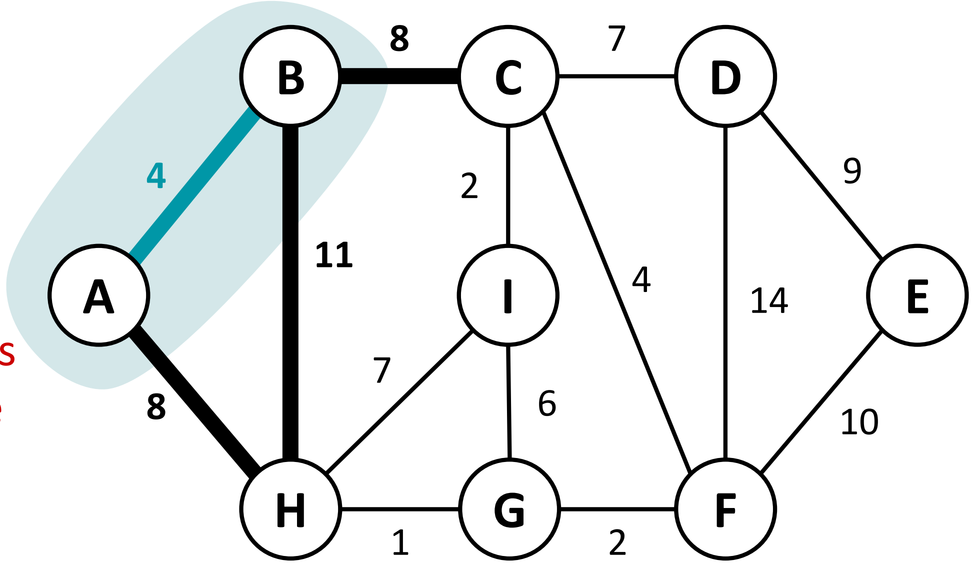
Our tree just grew by one! Now repeat until we reach all the nodes.

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.



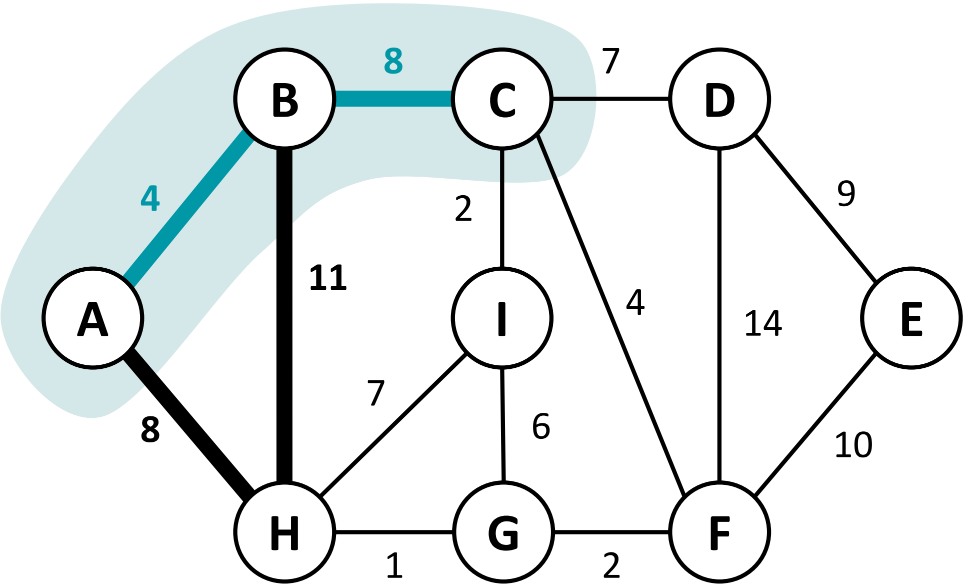
PRIM'S ALGORITHM: THE IDEA

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

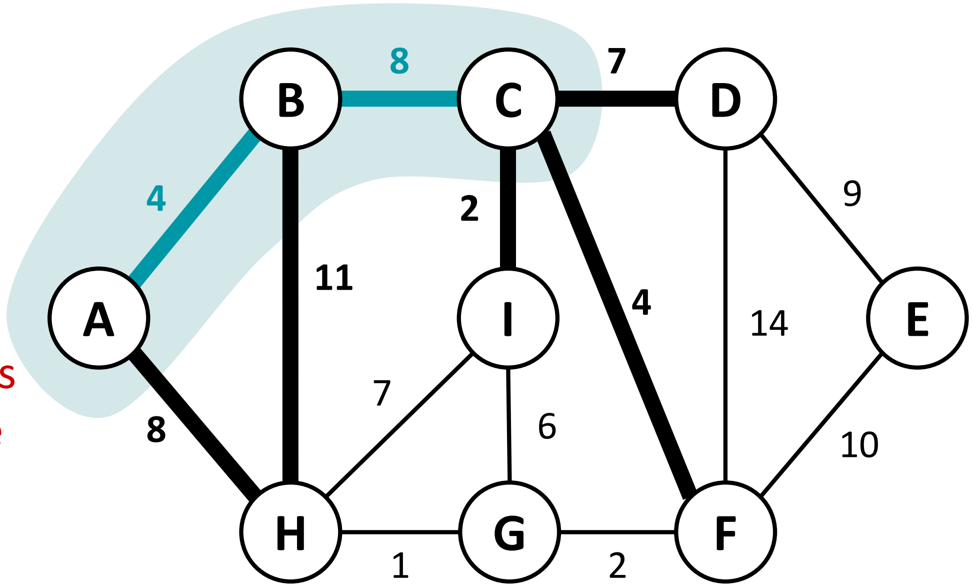
(if there's a tie, choose any)



PRIM'S ALGORITHM: THE IDEA

Greedy choice:

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



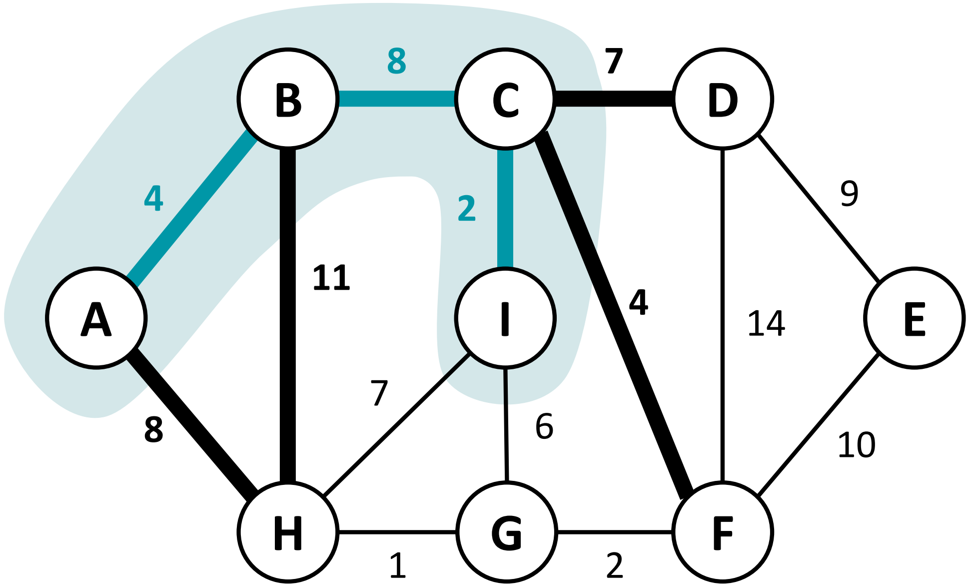
Consider the edges coming out of the "frontier" of our growing tree.

PRIM'S ALGORITHM: THE IDEA

Greedy choice:

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

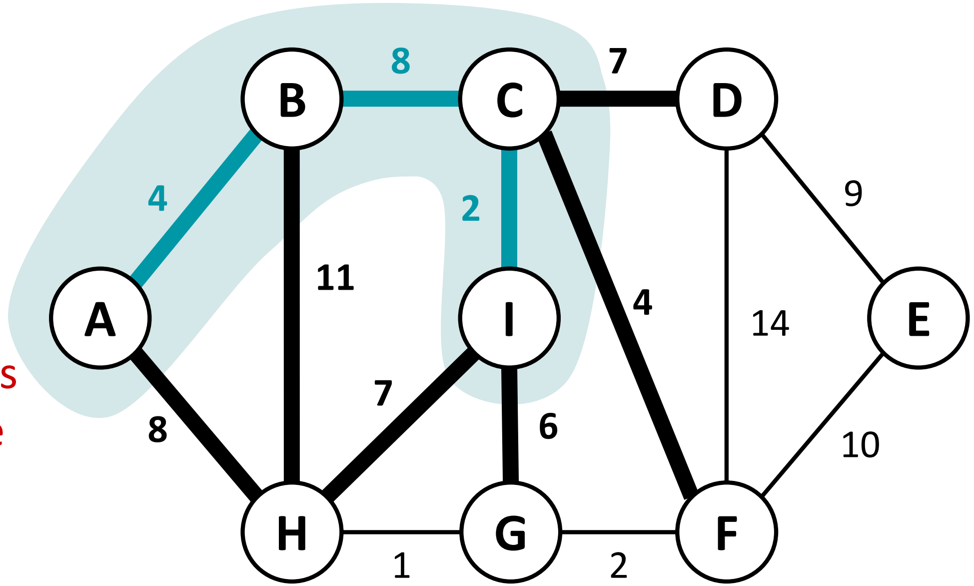
Claim the edge coming out of the "frontier" with the smallest weight



PRIM'S ALGORITHM: THE IDEA

Greedy choice:

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

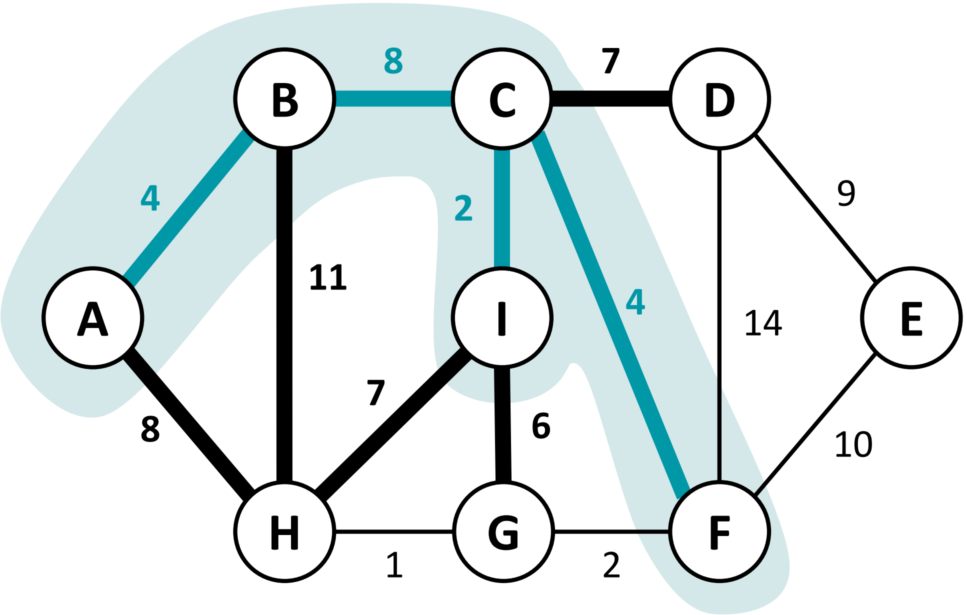


Consider the edges coming out of the "frontier" of our growing tree.

PRIM'S ALGORITHM: THE IDEA

Greedy choice:

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

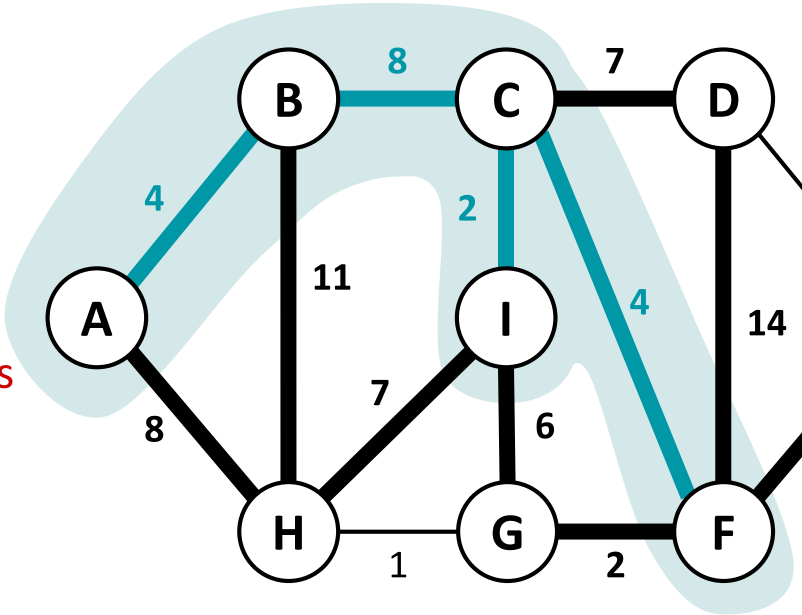


Claim the edge coming out of the "frontier" with the smallest weight

PRIM'S ALGORITHM: THE IDEA

Greedy choice:

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



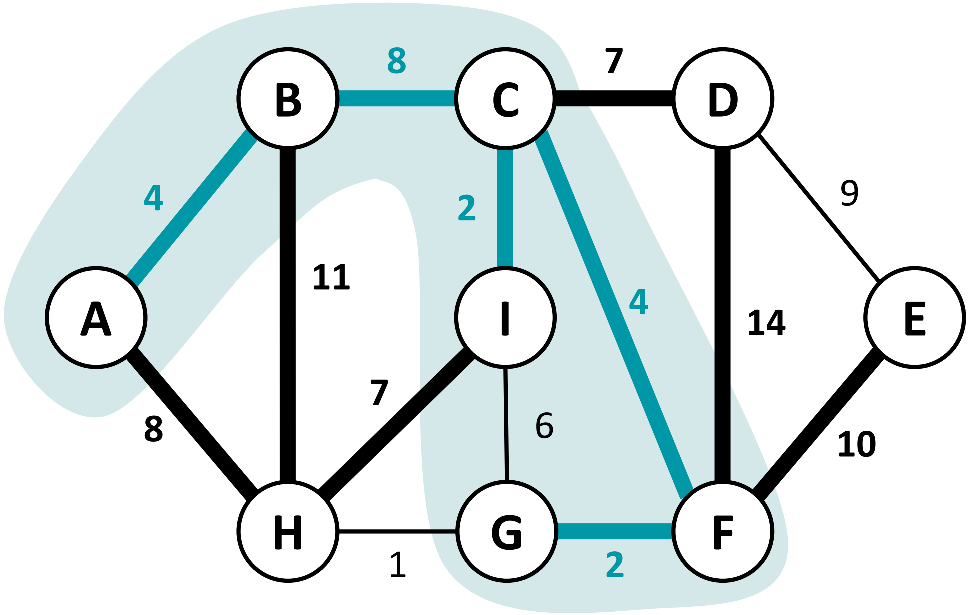
Consider the edges coming out of the "frontier" of our growing tree.

PRIM'S ALGORITHM: THE IDEA

Greedy choice:

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

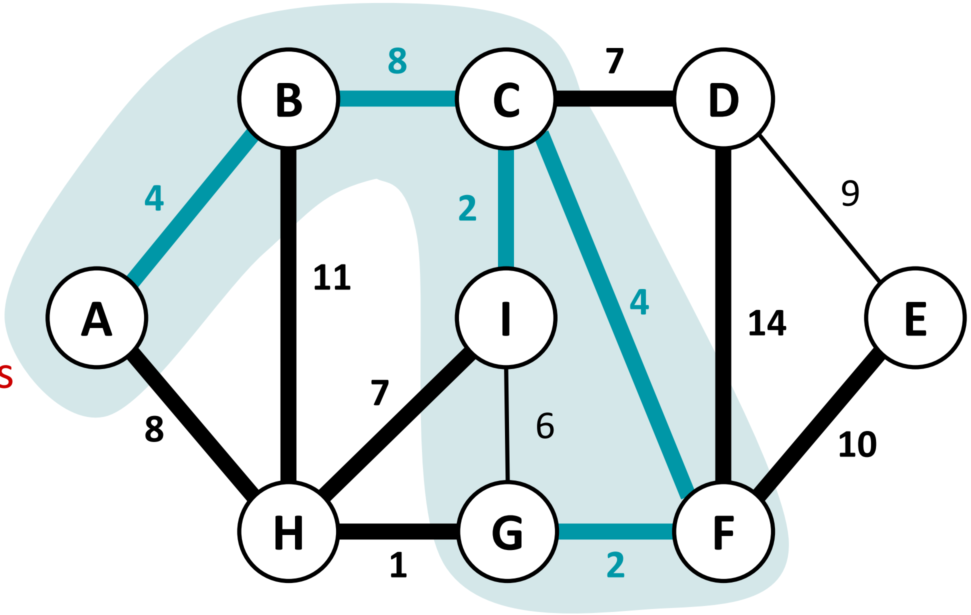
Claim the edge coming out of the "frontier" with the smallest weight



PRIM'S ALGORITHM: THE IDEA

Greedy choice:

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



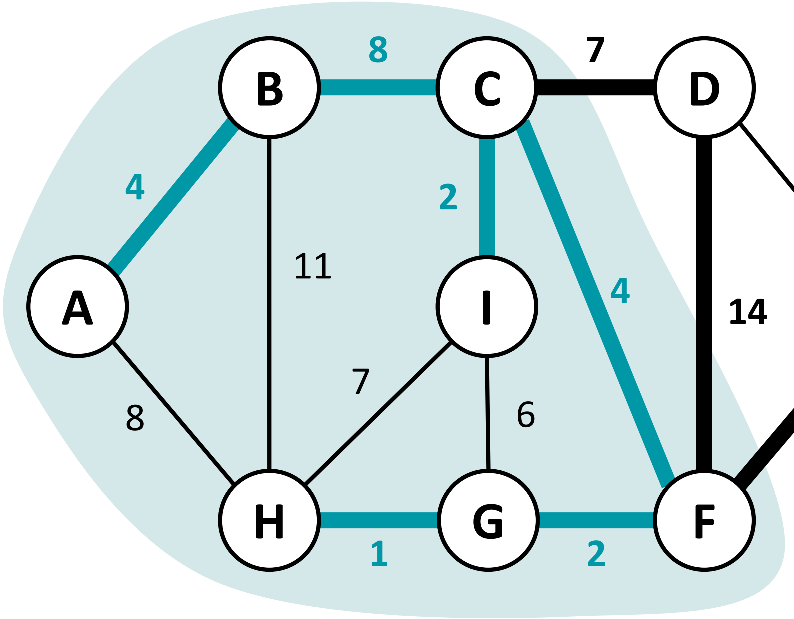
Consider the edges coming out of the "frontier" of our growing tree.

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

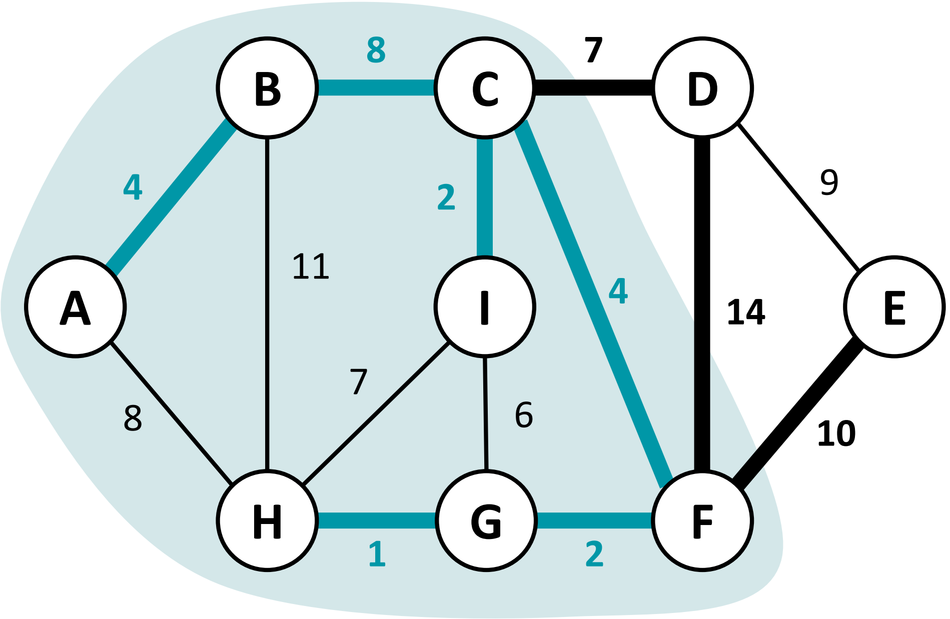


PRIM'S ALGORITHM: THE IDEA

Greedy choice:

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

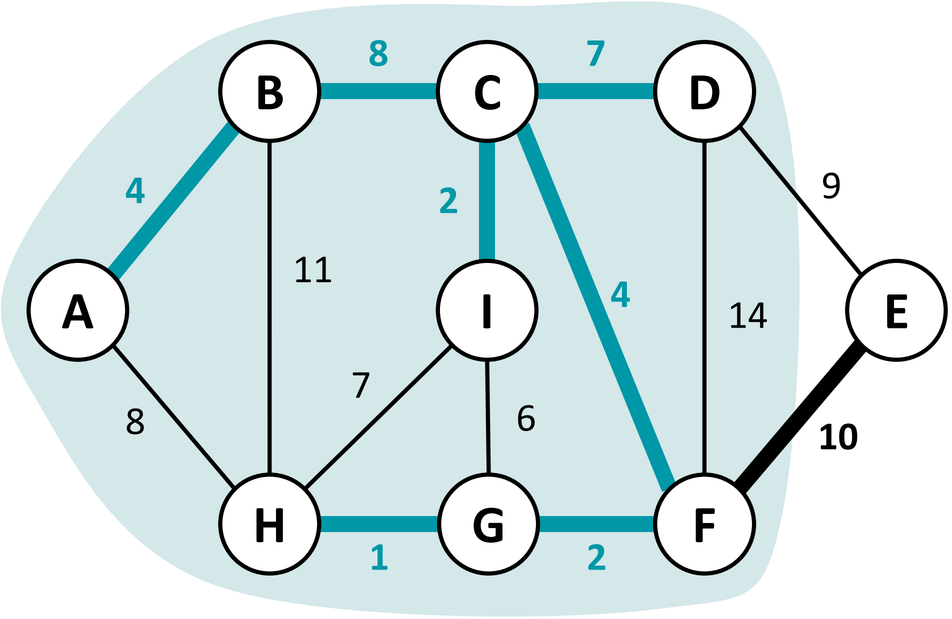
Consider the edges coming out of the "frontier" of our growing tree.



PRIM'S ALGORITHM: THE IDEA

Greedy choice:

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



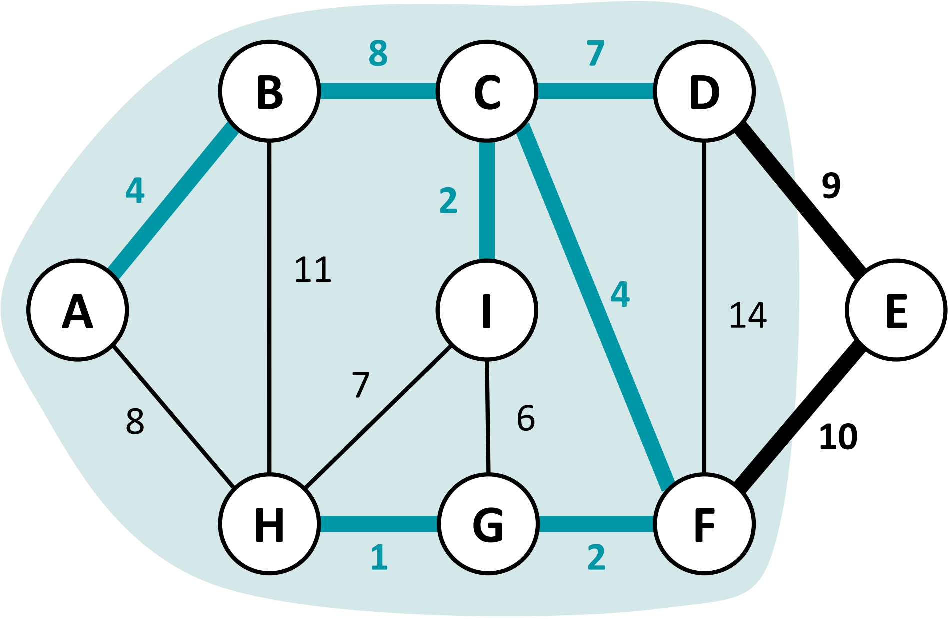
Claim the edge coming out of the "frontier" with the smallest weight

PRIM'S ALGORITHM: THE IDEA

Greedy choice:

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

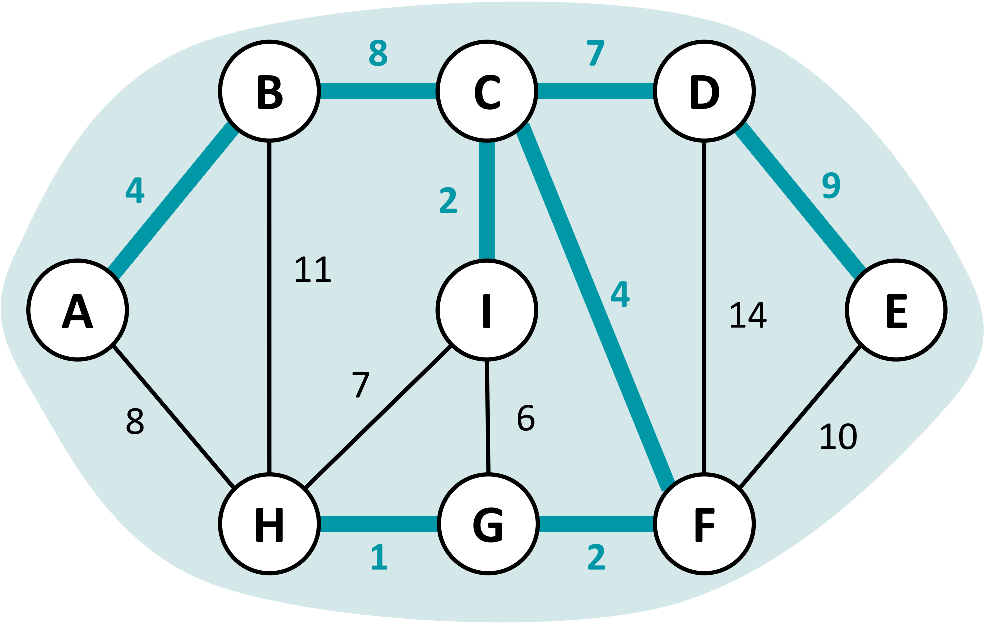
Consider the edges coming out of the "frontier" of our growing tree.



PRIM'S ALGORITHM: THE IDEA

Greedy choice:

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

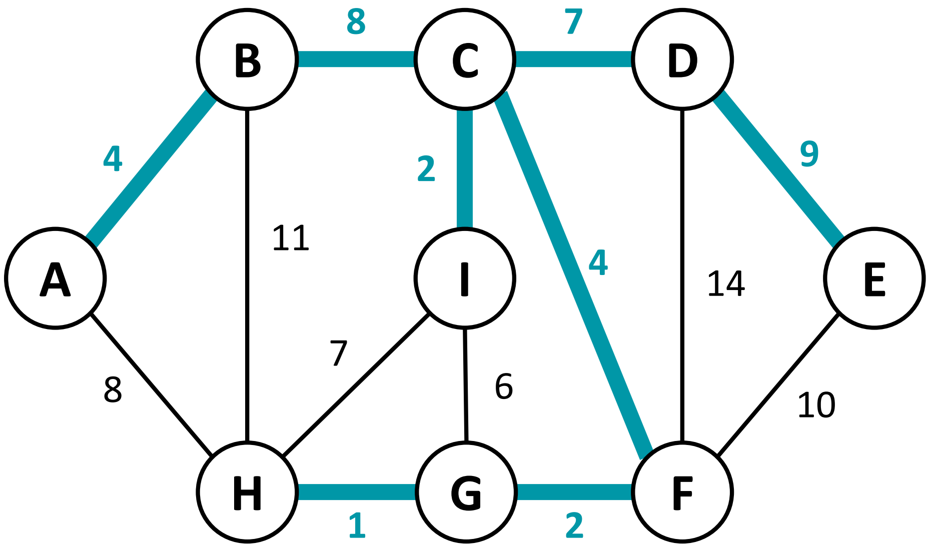


Claim the edge coming out of the "frontier" with the smallest weight

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(E) time per iteration..

(Naive) Runtime: O(V . E)

(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)

the

ch

O(E)

on..

return MST

(Naive) Runtime: O(V . E)

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

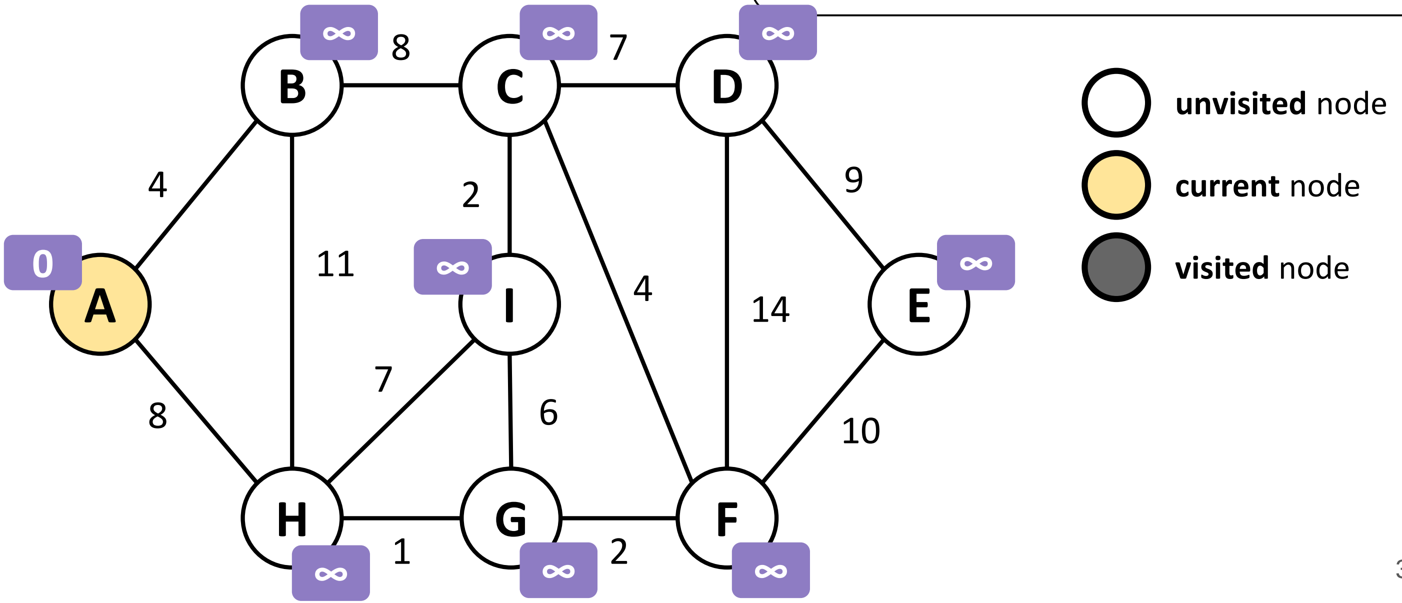
HOW DO WE IMPLEMENT THIS?

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)

PRIM(G = (V,E), s):
  MST = {}
  visited = {s}
  for all v besides s: **d[v] = ∞** and **k[v] = NULL**

A is part of the growing tree first

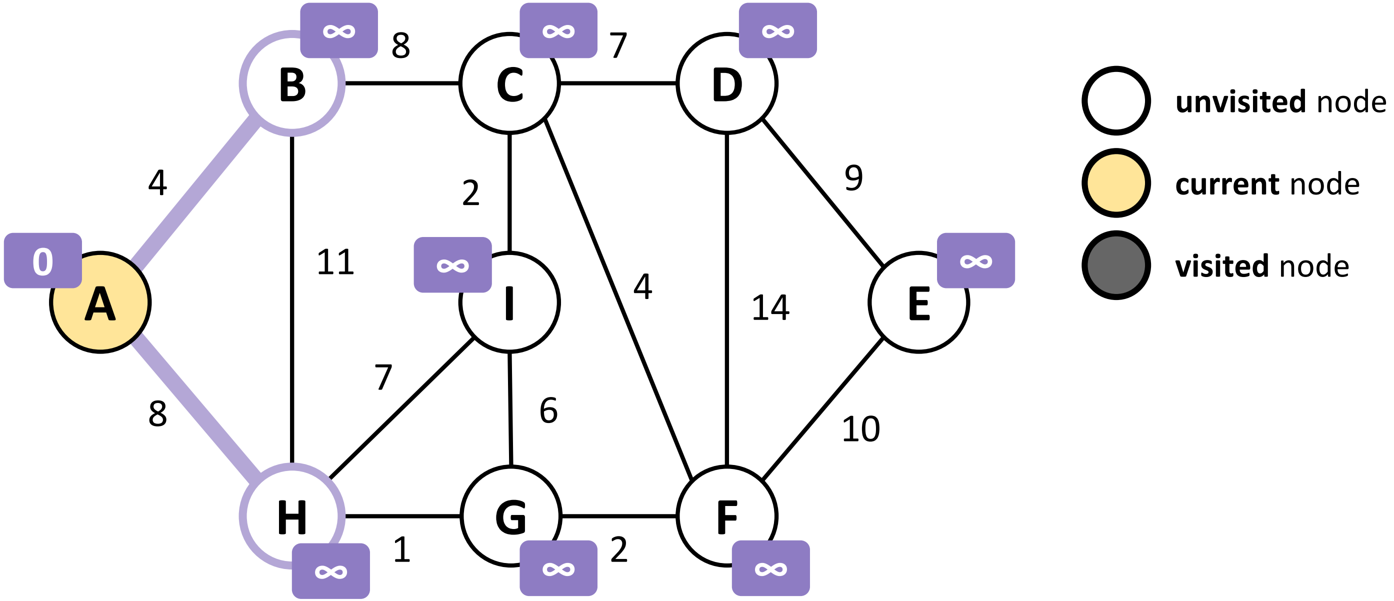


HOW DO WE IMPLEMENT THIS?

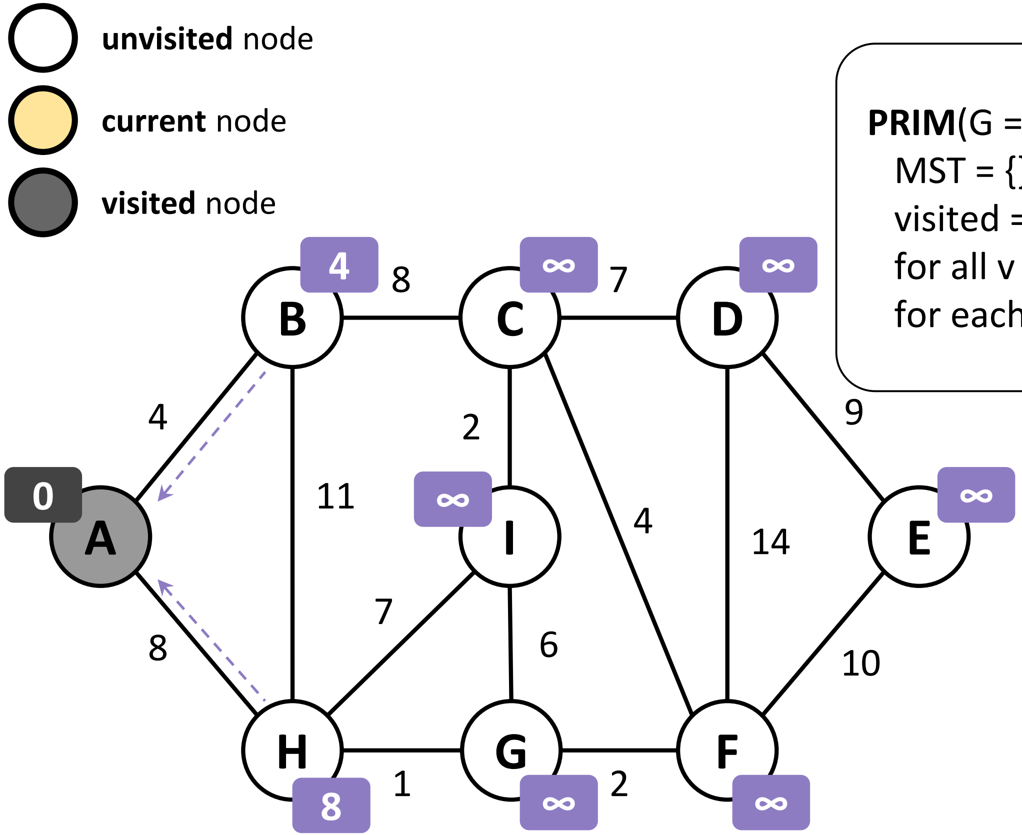
Each vertex that's not yet reached by the growing tree keeps track of:

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?

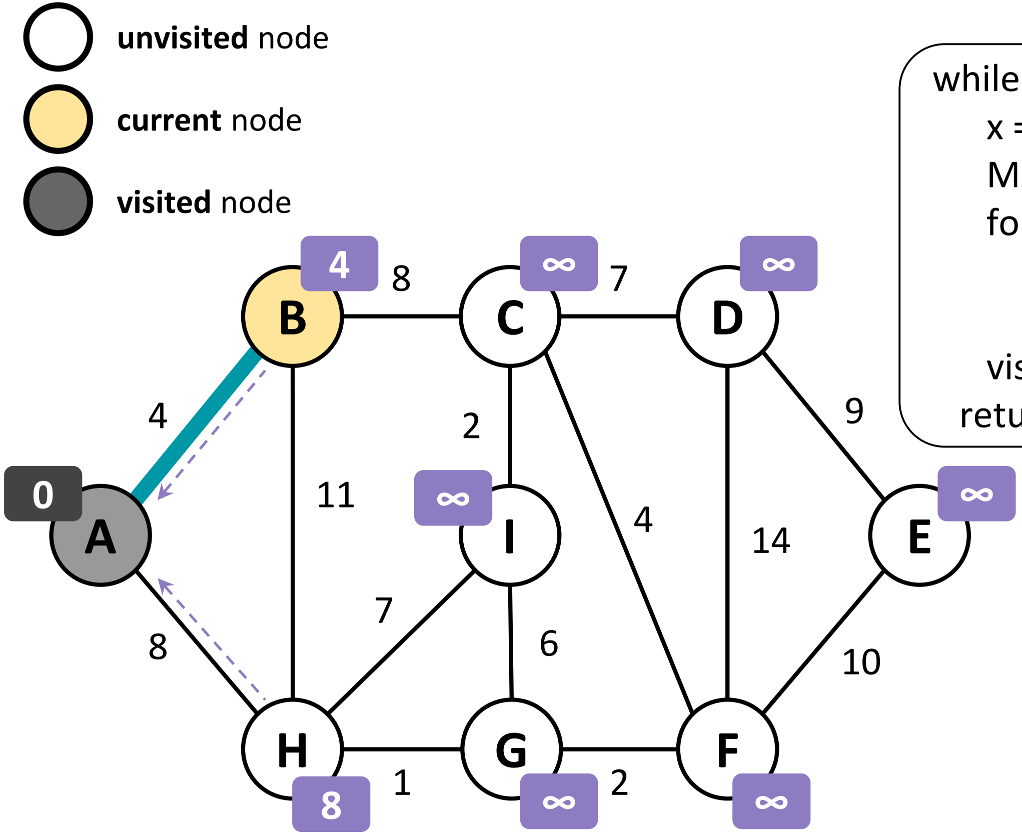


PRIM(G = (V,E), s):
  MST = {}
  visited = {s}
  for all v besides s: **d[v] = ∞** and **k[v] = NULL**
  for each neighbor v of s: **d[v]** = **w(s,v)** and **k[v]** = **s**

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)

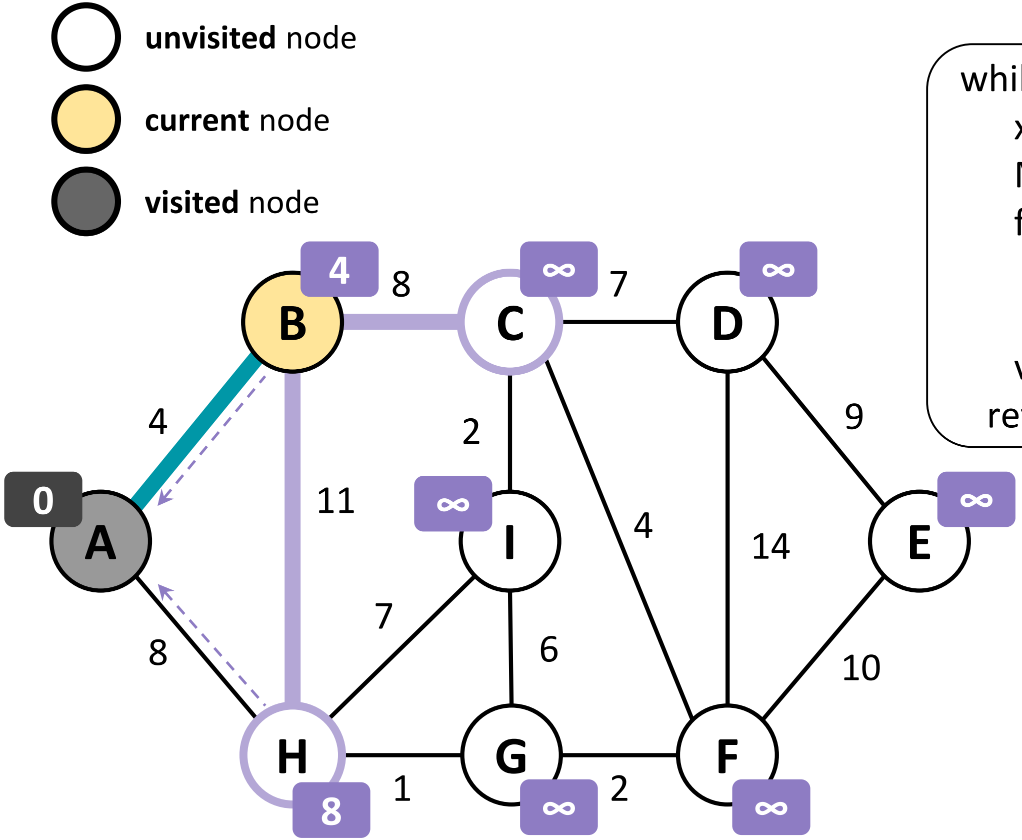
HOW DO WE IMPLEMENT THIS?




```
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
```

B is the closest node to the growing tree.

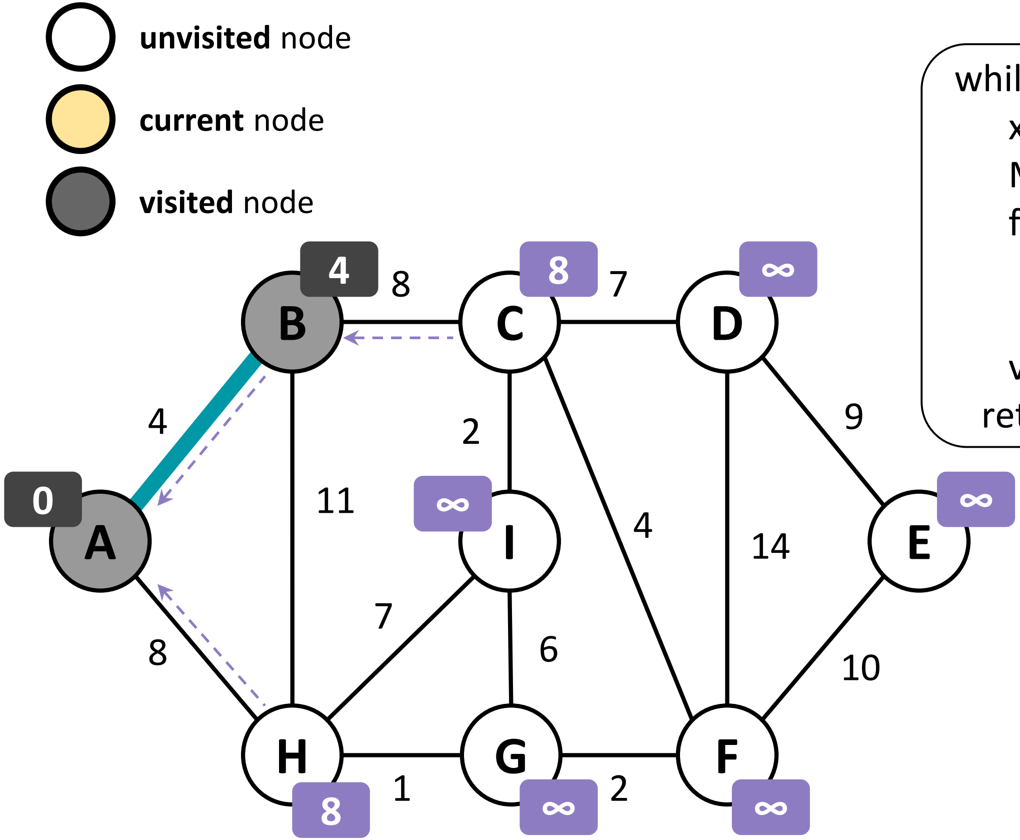
HOW DO WE IMPLEMENT THIS?




```
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
```

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?




```
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
```

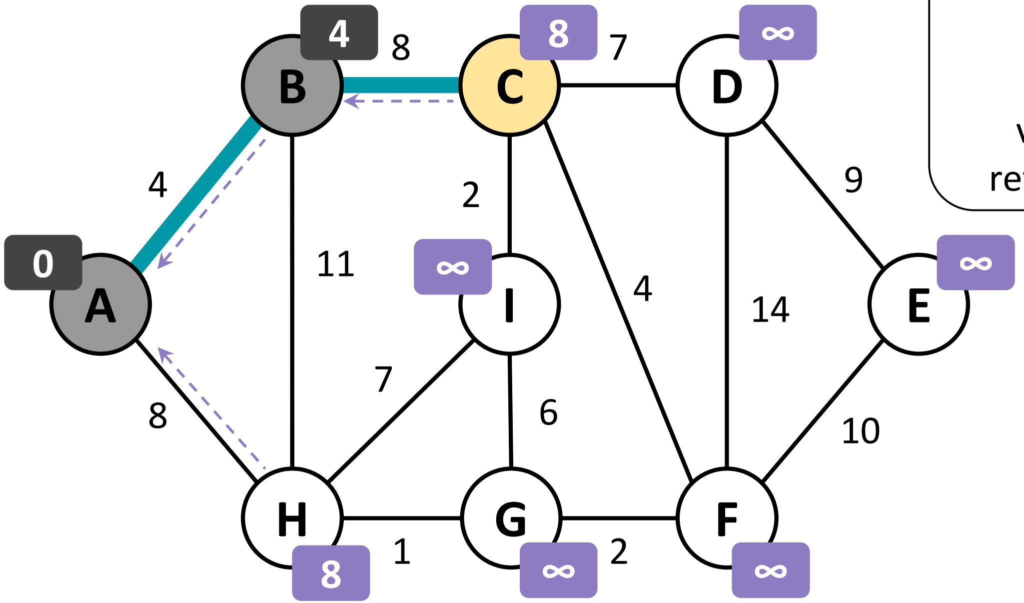
Update their estimates, and now B is officially done.

HOW DO WE IMPLEMENT THIS?

unvisited node

current node

visited node




```
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
```

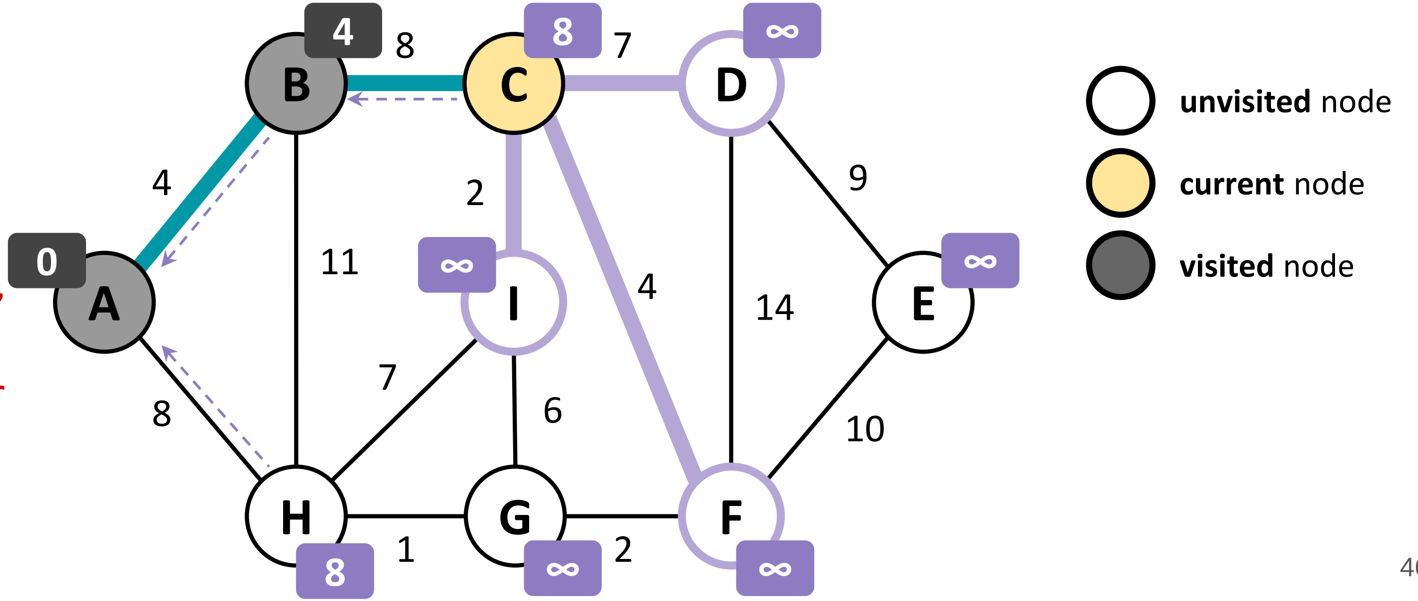
C is the closest node to the growing tree.

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!



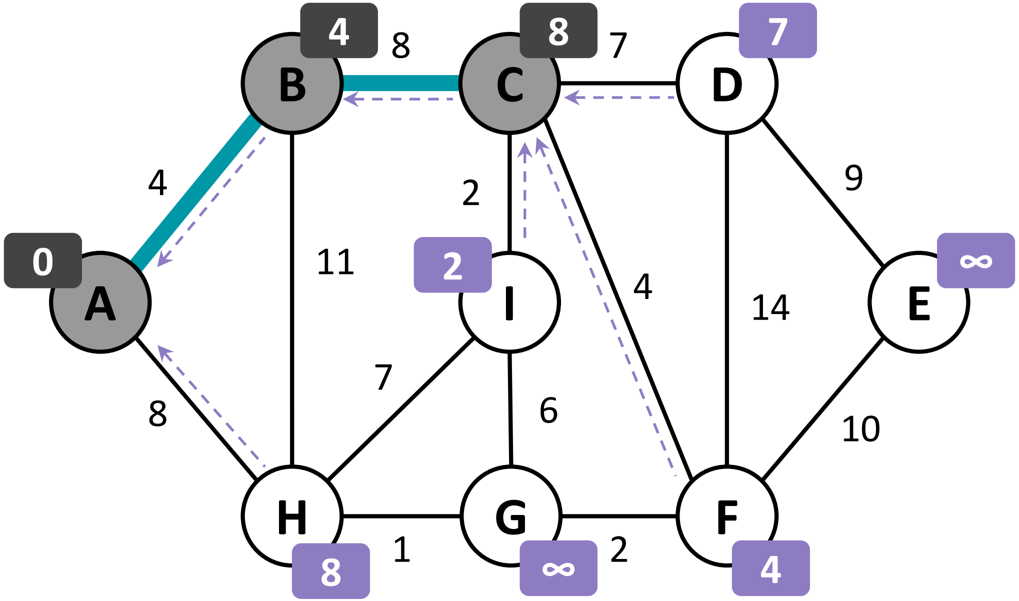
HOW DO WE IMPLEMENT THIS?

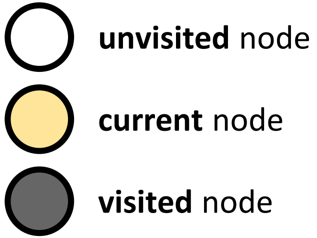
Each vertex that's not yet reached by the growing tree keeps track of:

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)





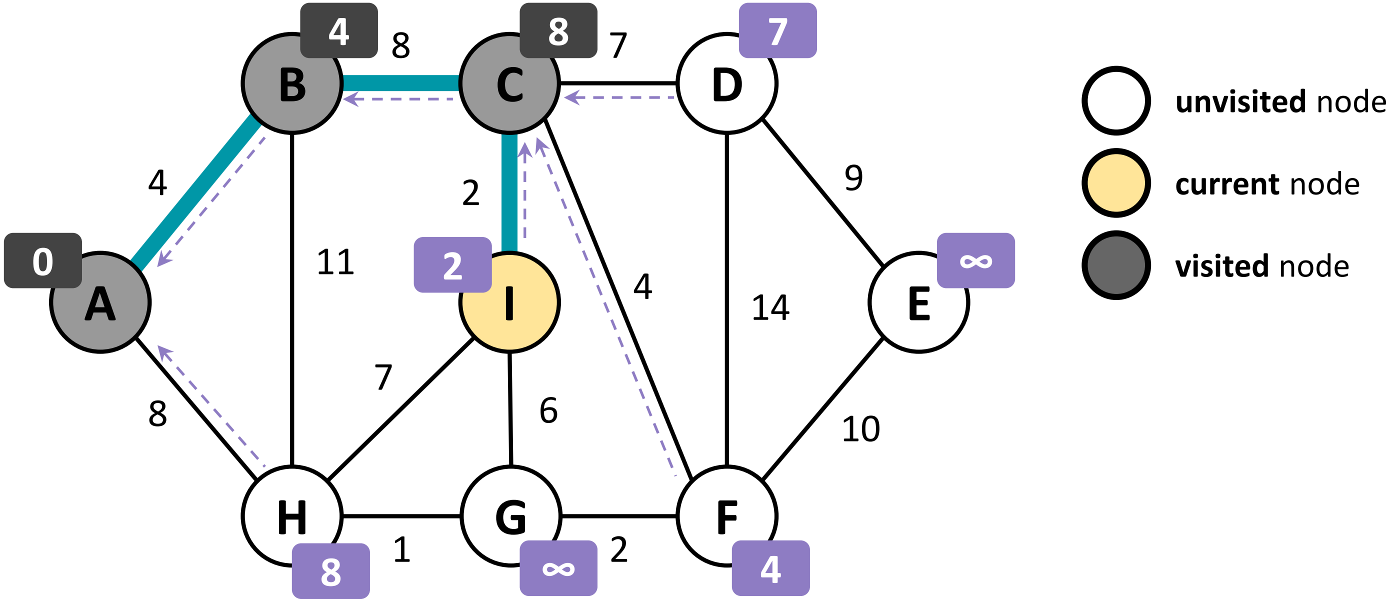
HOW DO WE IMPLEMENT THIS?

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.

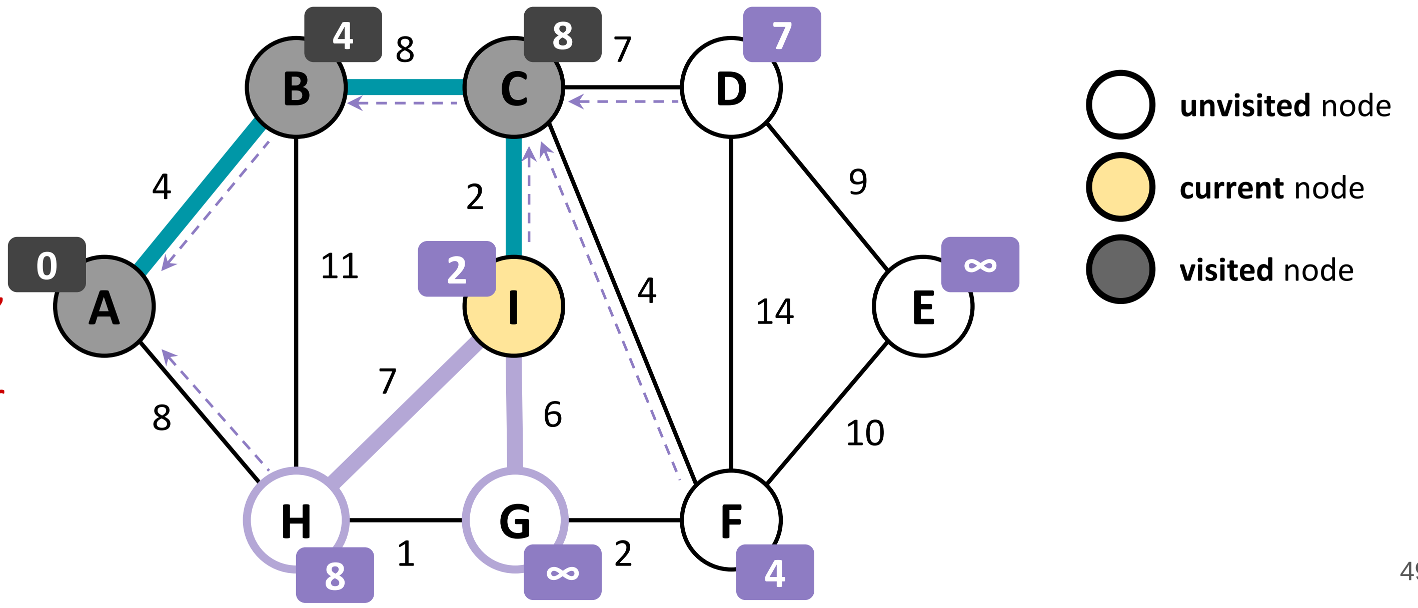


HOW DO WE IMPLEMENT THIS?

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!



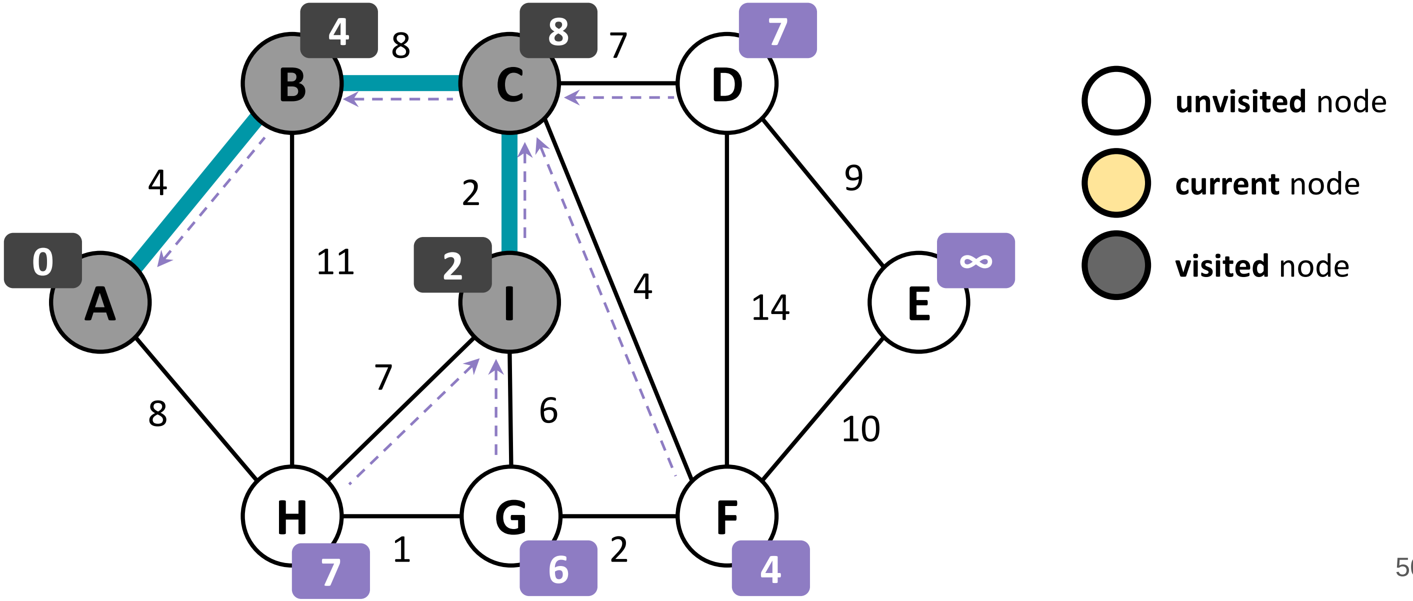
HOW DO WE IMPLEMENT THIS?

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

```
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(E) time per iteration..

(Naive) Runtime: O(V . E)

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

```
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
```

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 Min-heap): O(E log V)

HEAPSORT(*A*)

From Previous Lecture Slides

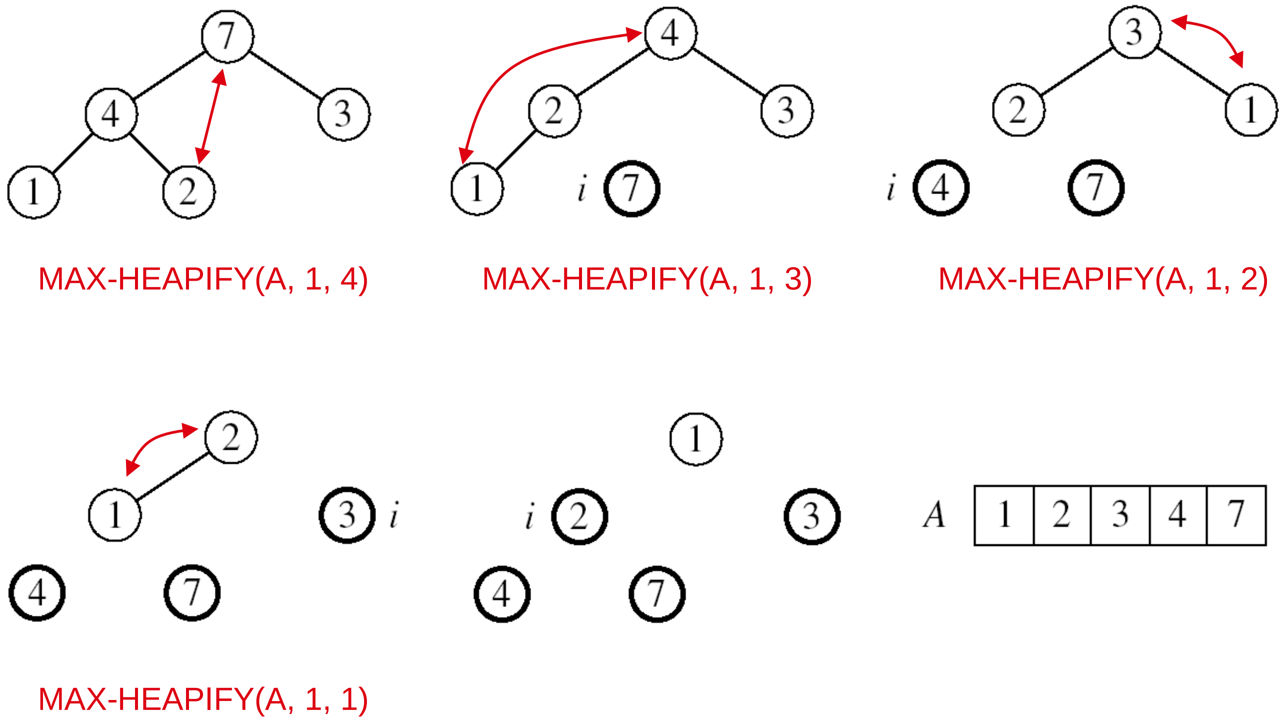
1. BUILD-MAX-HEAP(A)
2. **for** i ← $length[A]$ **downto** 2
3. **do** exchange $A[1] \boxed{\leftrightarrow} A[i]$
4. MAX-HEAPIFY(A, 1, i - 1)

$O(\lg n)$ } n-1 times

- Running time: $O(n\lg n)$ --- Can be shown to be $\Theta(n\lg n)$

Example: A=[7, 4, 3, 1, 2]

From Previous Lecture Slides



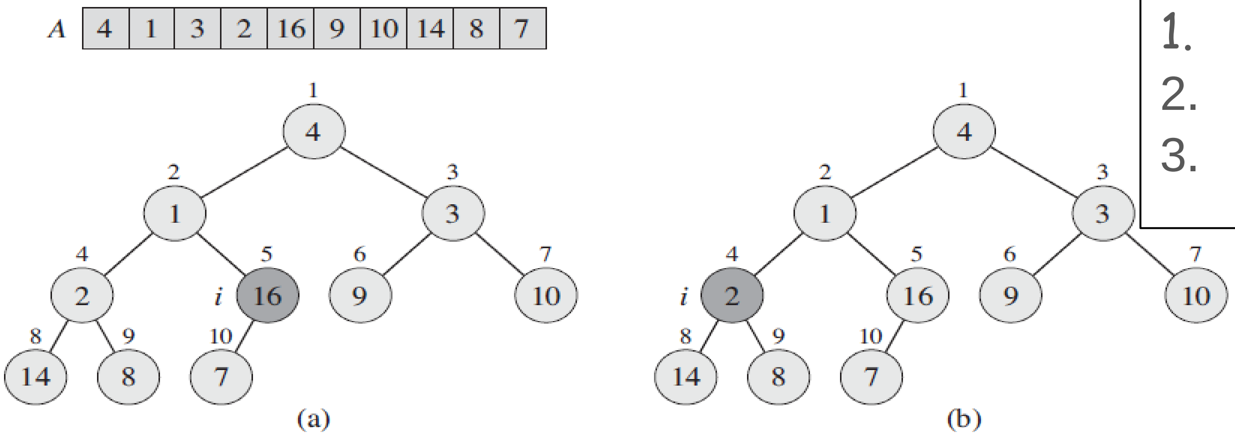
A	1	2	3	4	7

Build Max Heap Procedure

From Previous Lecture Slides

- Convert an array $A[1 \ldots n]$ into a max-heap (n = $length[A]$)
- The elements in the subarray $A[(\lfloor n/2 \rfloor+1) \ldots n]$ are leaves
- Apply MAX-HEAPIFY on elements between 1 and $\lfloor n/2 \rfloor$

Figure 6.3 :



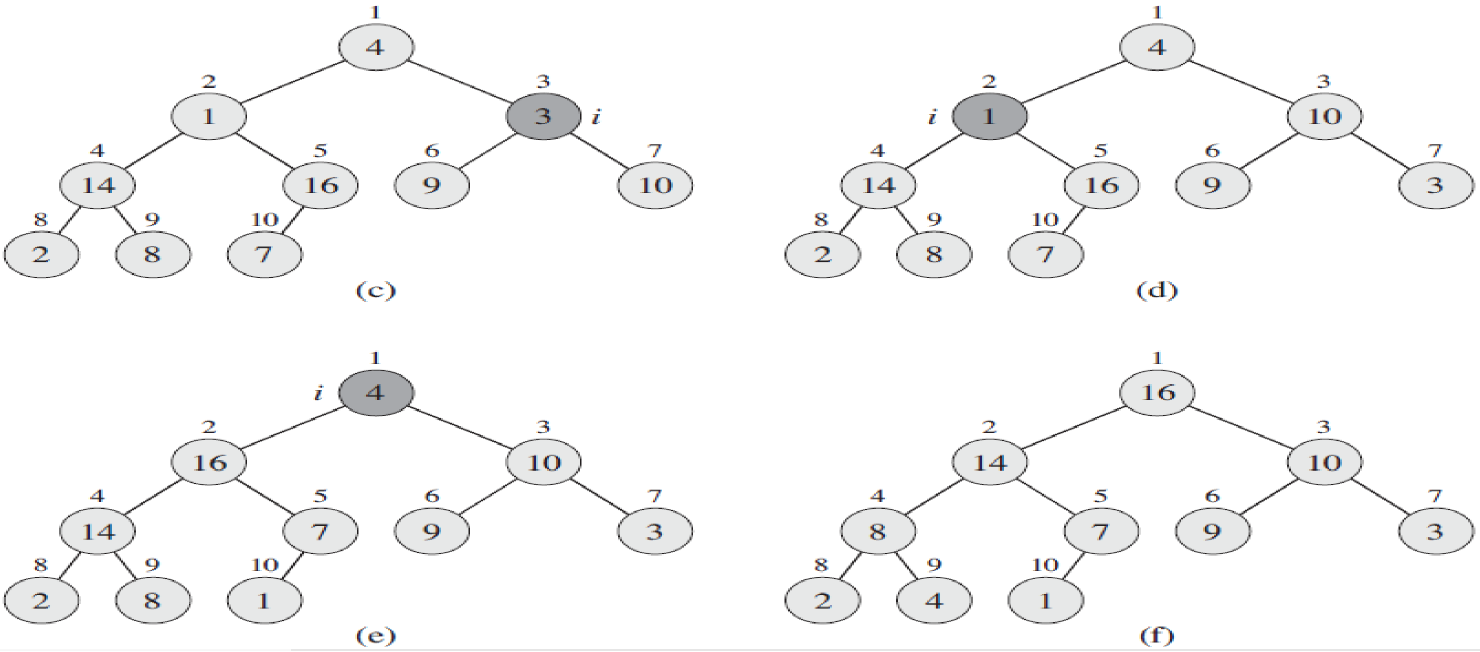
BUILD-MAX-HEAP(*A*)

```
n = length[A]
for i ← ⌊n/2⌋ downto 1
    do MAX-HEAPIFY(A, i, n)
```

Build Max Heap Procedure

From Previous Lecture Slides

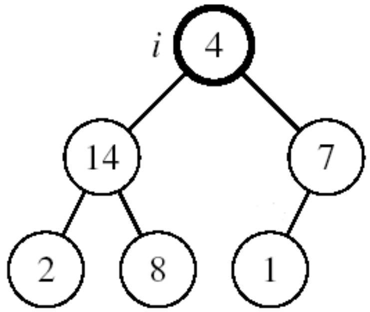
- Figure 6.3 :



Maintaining the Heap Property

From Previous Lecture Slides

- Assumptions:
  - Left and Right subtrees of *i* are max-heaps
  - *A*[i] may be smaller than its children



MAX-HEAPIFY(A, i, n)

```
l ← LEFT(i)
r ← RIGHT(i)
if l ≤ n and A[l] > A[i]
    then largest ←l
    else largest ←i
if r ≤ n and A[r] > A[largest]
    then largest ←r
if largest ≠ i
    then exchange A[i] ↔ A[largest]
        MAX-HEAPIFY(A, largest, n)
```

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

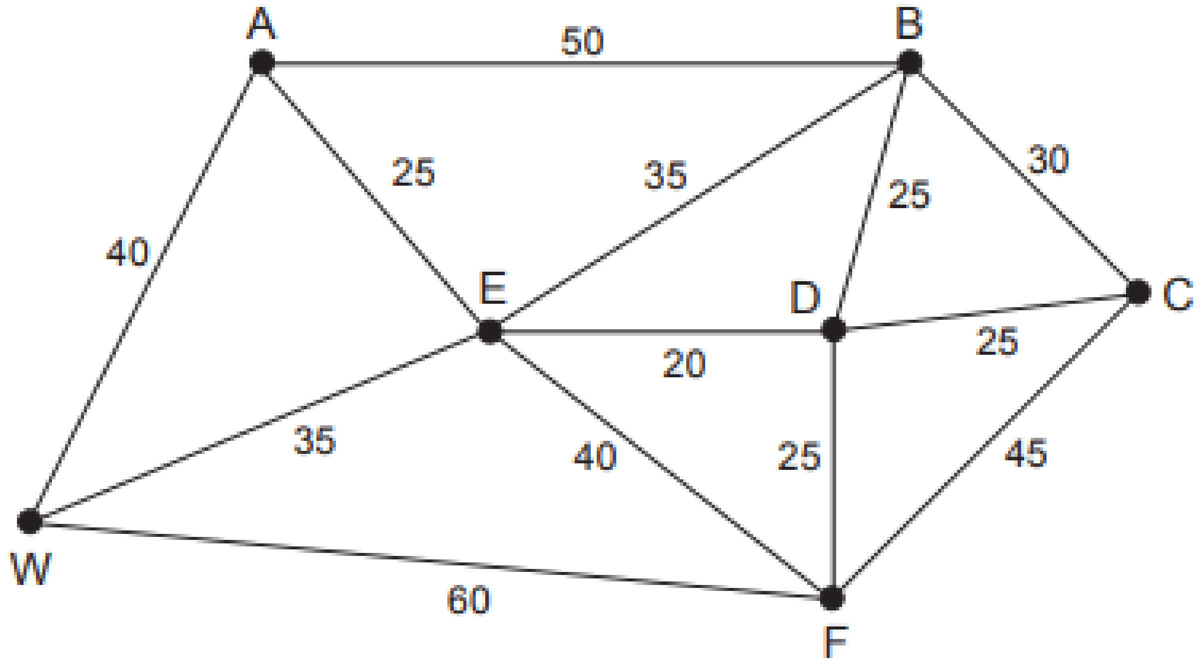
Find clusters in a dataset (one of the algorithms we'll see today can be modified slightly to basically do this)

Useful primitive

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

PRIM'S ALGORITHM: VERSION

Travel Agency wants to setup a public transport system between all the cities. The passenger fare in rupees between the cities are shown in the **Figure-2**. How should all cities be linked to maximize the total fare. [Hint: Use Spanning Tree]



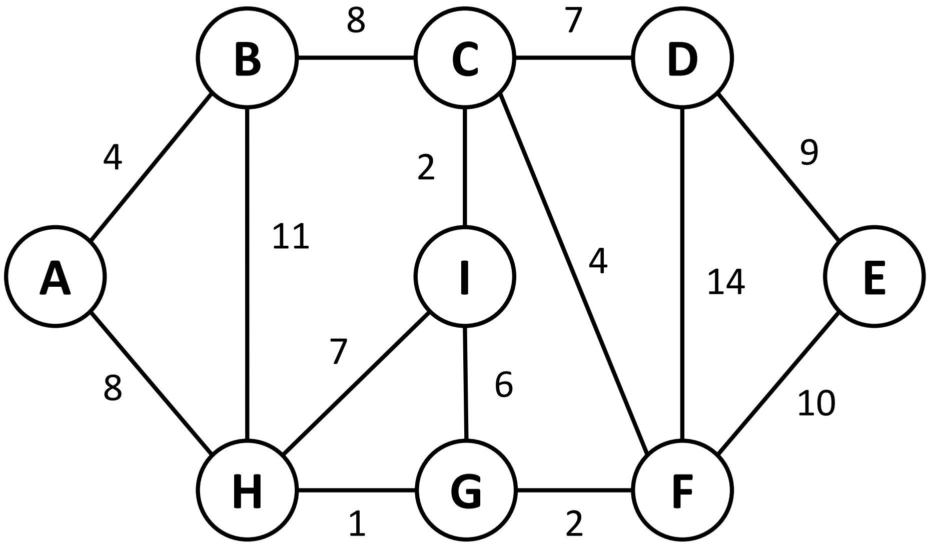
KRUSKAL'S ALGORITHM

Greedily add the cheapest edge!

KRUSKAL'S ALGORITHM: THE IDEA

Greedy choice:

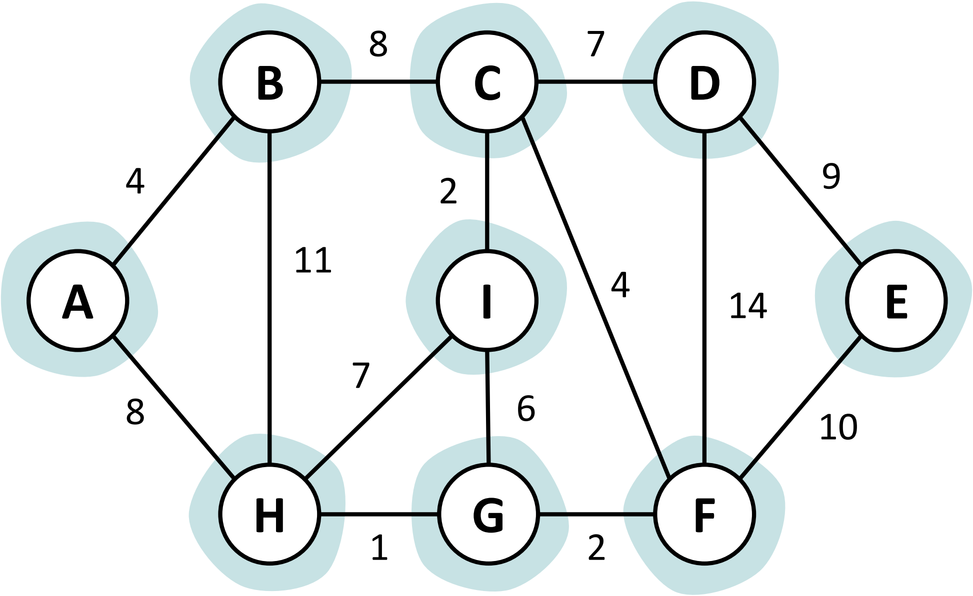
Maintain a forest of trees, & greedily add the cheapest edge to combine trees



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

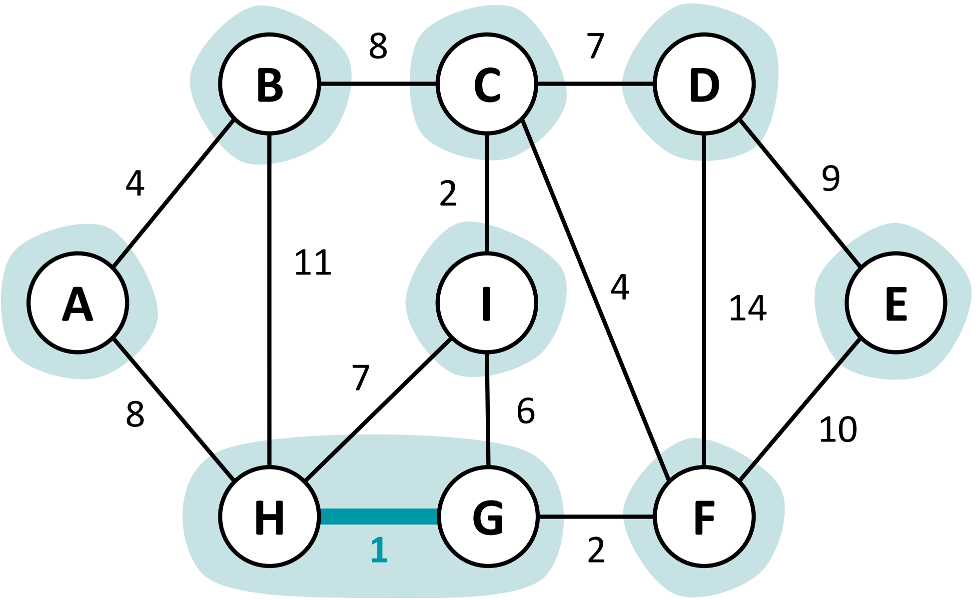
KRUSKAL'S ALGORITHM: THE IDEA

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)



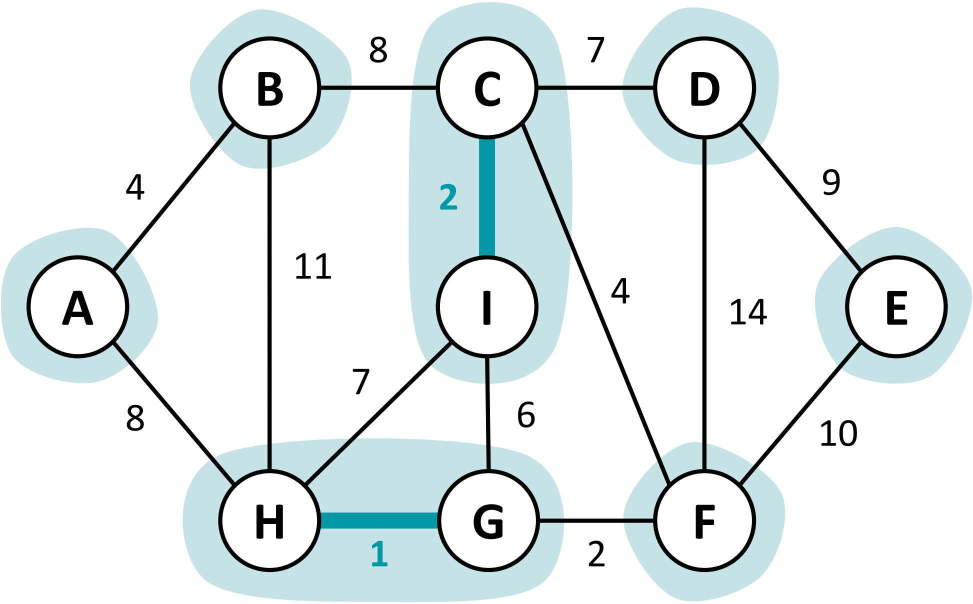
KRUSKAL'S ALGORITHM: THE IDEA

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

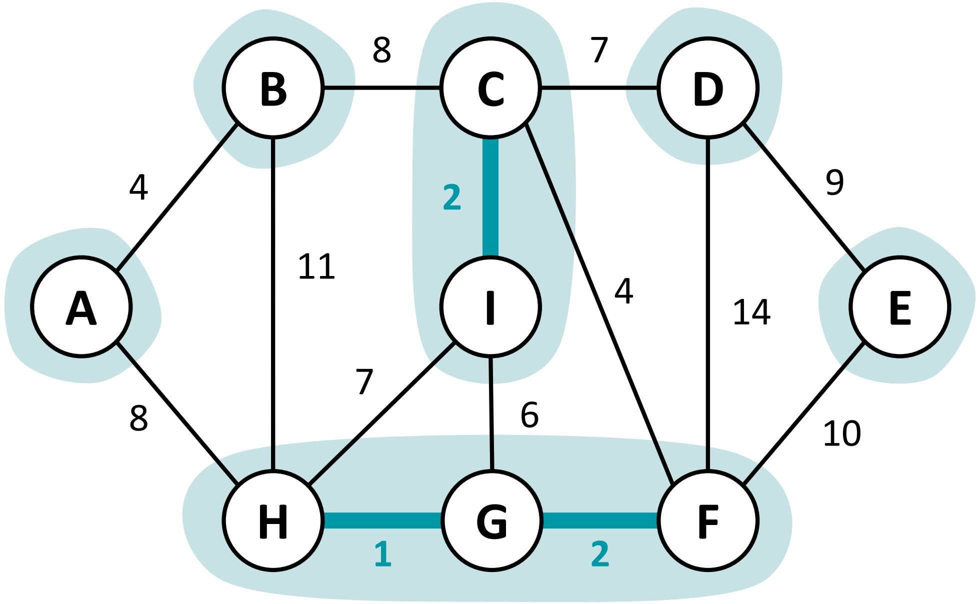
KRUSKAL'S ALGORITHM: THE IDEA

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)



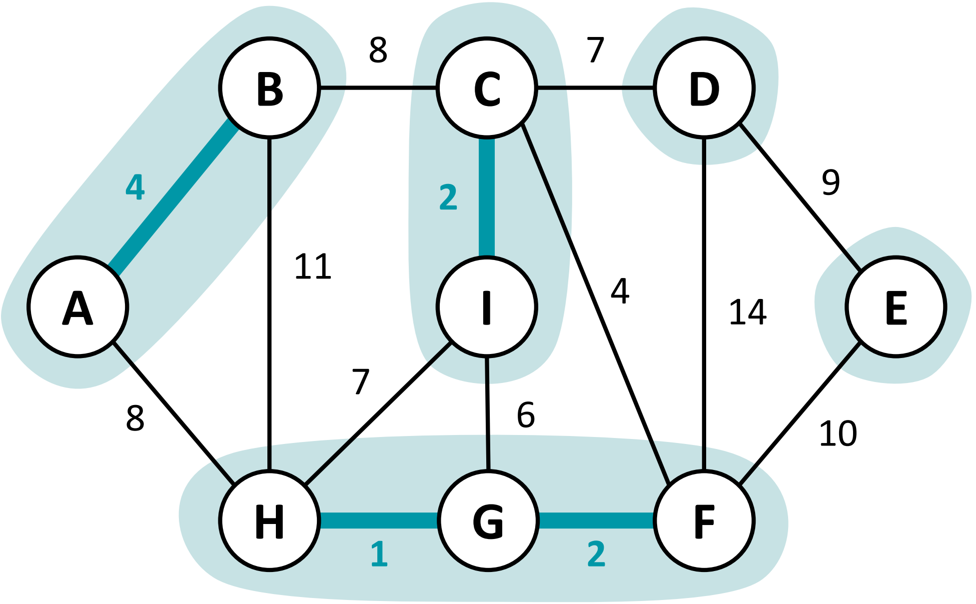
KRUSKAL'S ALGORITHM: THE IDEA

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

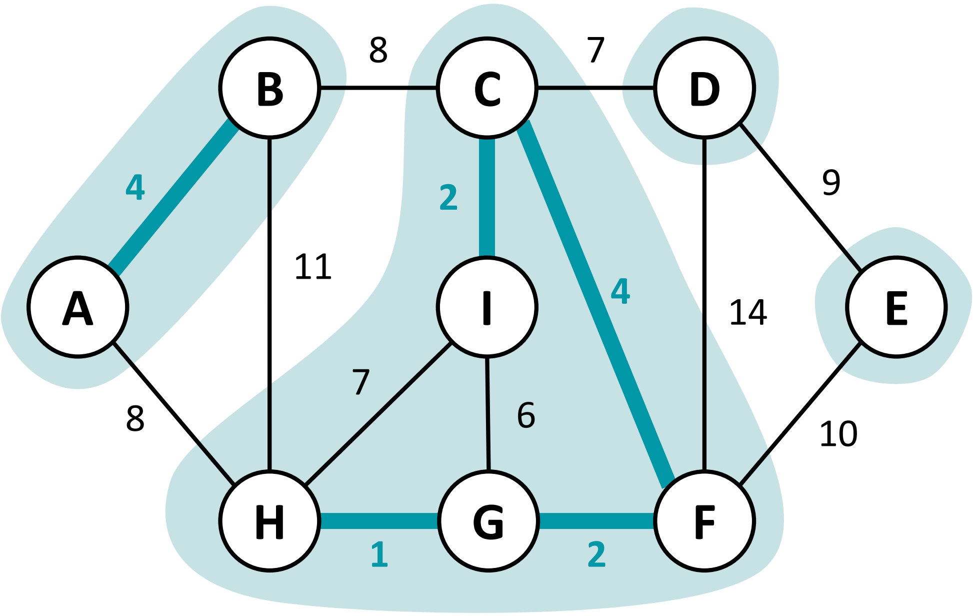
KRUSKAL'S ALGORITHM: THE IDEA

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)



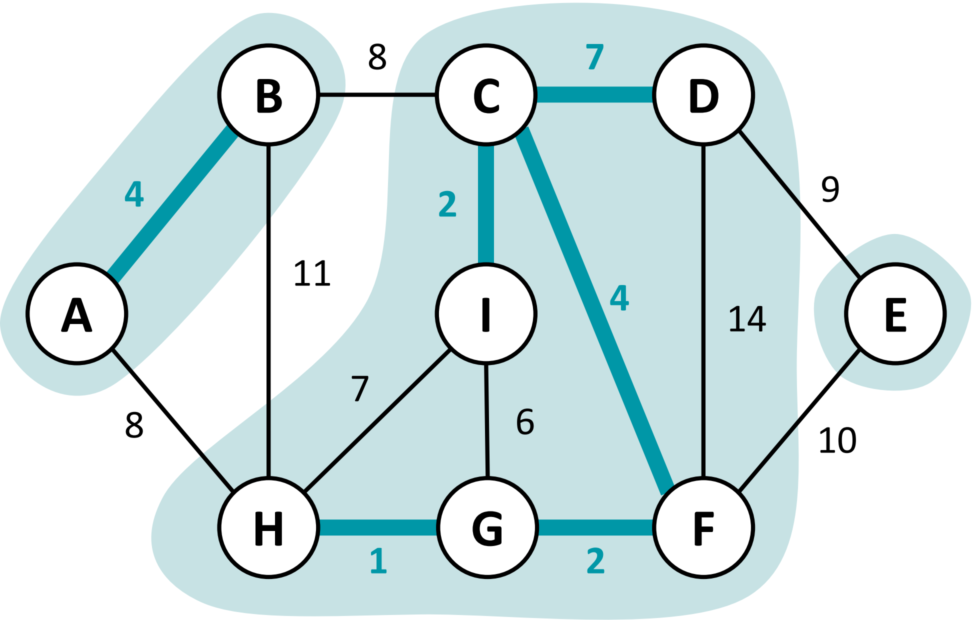
KRUSKAL'S ALGORITHM: THE IDEA

Greedy choice:

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

Choose the cheapest edge that would combine two trees

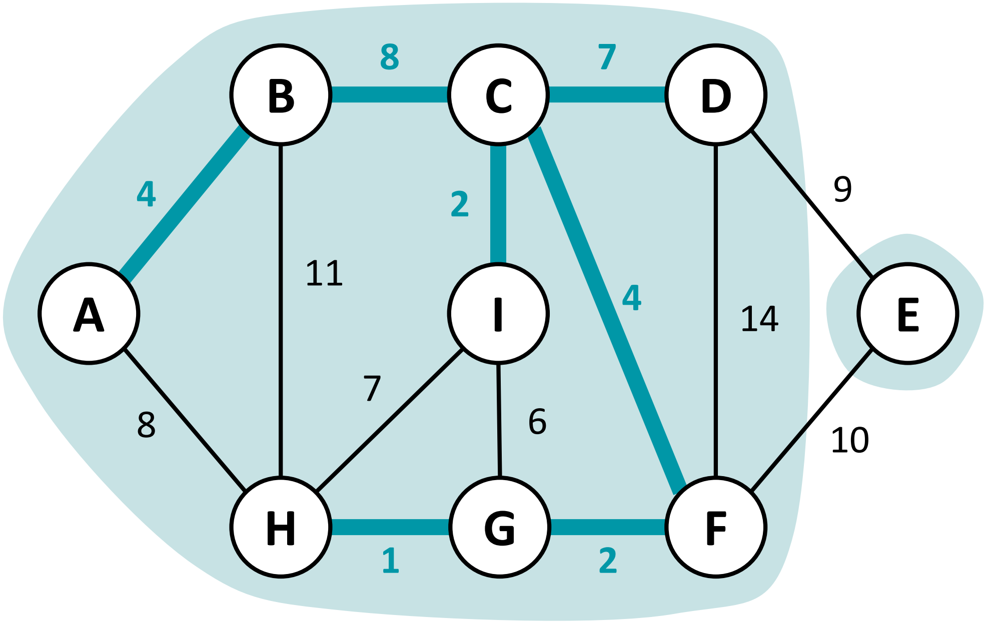
(i.e. that won't cause a cycle)



KRUSKAL'S ALGORITHM: THE IDEA

Greedy choice:

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



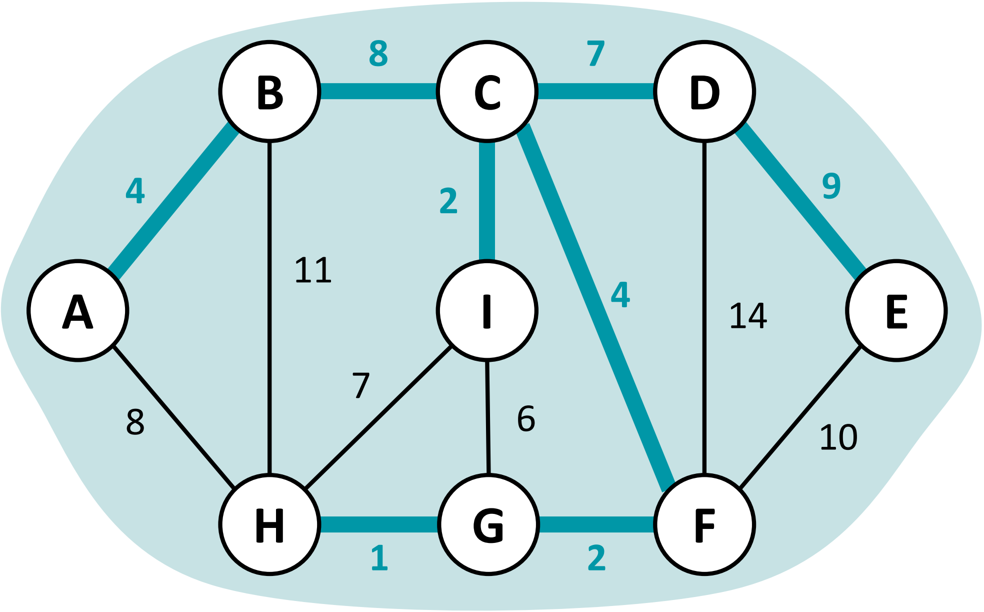
Choose the cheapest edge that would combine two trees

(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

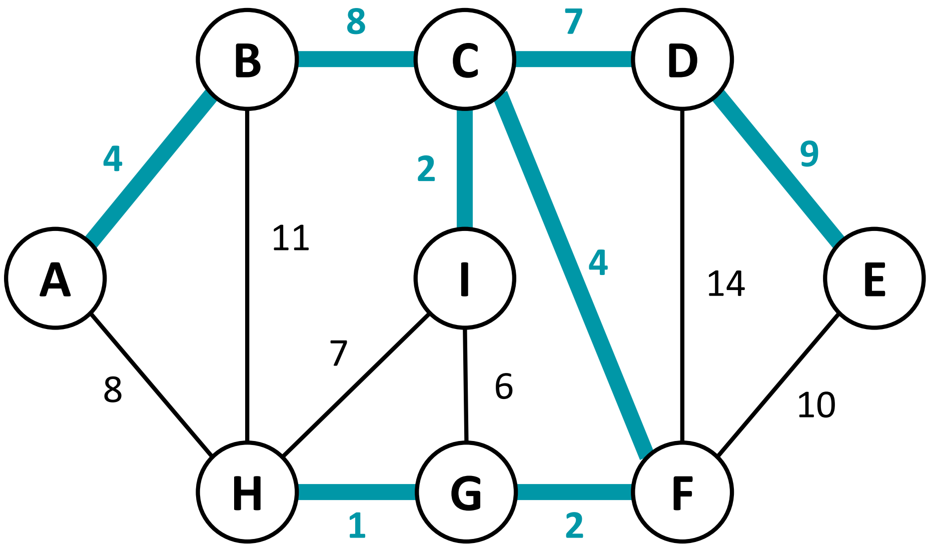
(i.e. that won't cause a cycle)

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

```
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
```

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

(With union-find data structure) **Runtime = $O(E \log V)$**

CLRS textbook version PSEUDOCODE For KRUSKAL'S ALGORITHM

```
MST-KRUSKAL(G, w)
1  A = ∅
2  for each vertex v ∈ G.V
3      MAKE-SET(v)
4  sort the edges of G.E into nondecreasing order by weight w
5  for each edge (u, v) ∈ G.E, taken in nondecreasing order by weight
6      if FIND-SET(u) ≠ FIND-SET(v)
7          A = A ∪ {(u, v)}
8          UNION(u, v)
9  return A
```

since $E \le V^2$, we have log E = O (log V)

O(E log E) = O(E log V),

Runtime (Time to sort line 4): O(E log E) (merge sort)

(Make Set |V|, for loop 5-8 : O (E)

Total Algo Runtime = O (E log E)