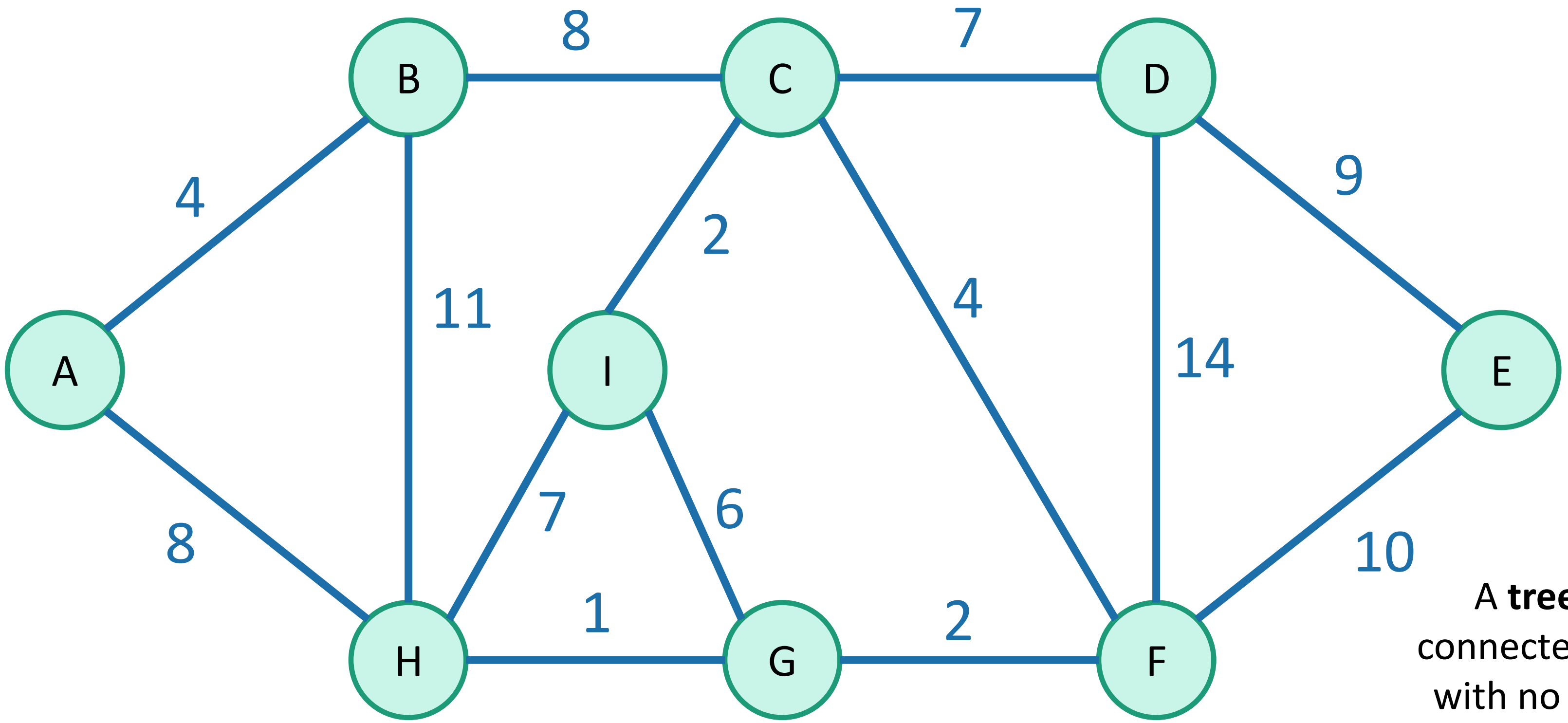
Minimum Spanning Trees

Minimum Spanning Tree

Say we have an undirected weighted graph



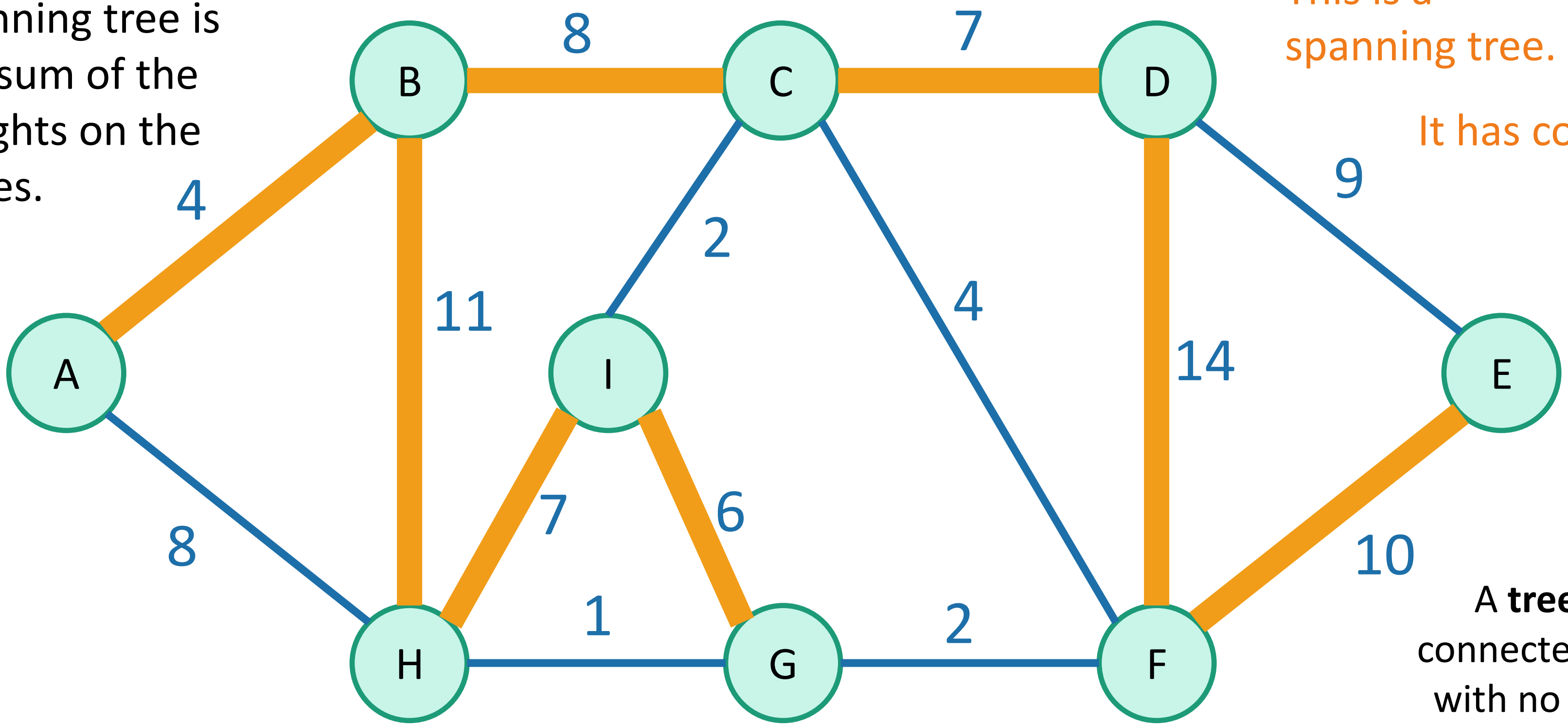
A **tree** is a connected graph with no cycles!

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

Minimum Spanning Tree

Say we have an undirected weighted graph

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



This is a spanning tree.

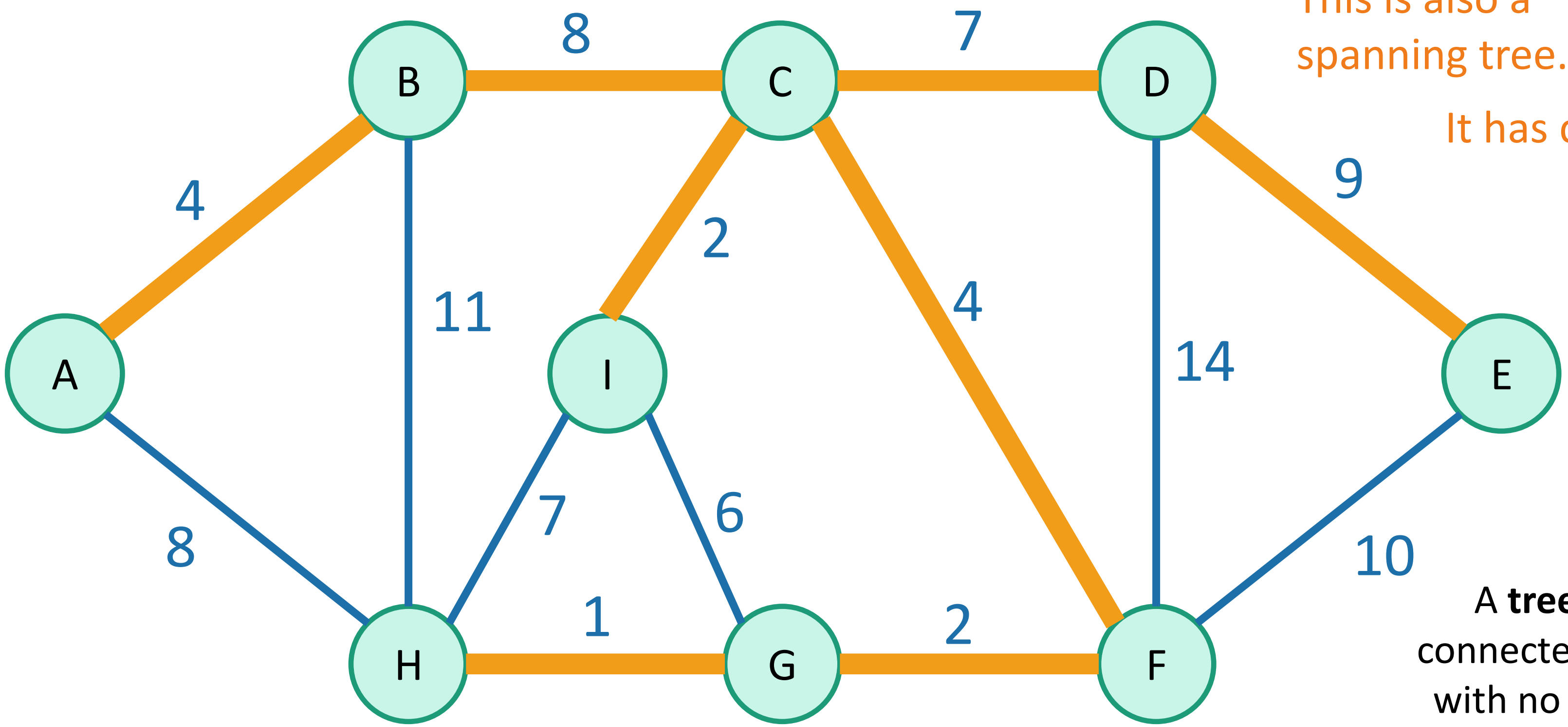
It has cost 67

A **tree** is a connected graph with no cycles!

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

Minimum Spanning Tree

Say we have an undirected weighted graph



This is also a spanning tree.

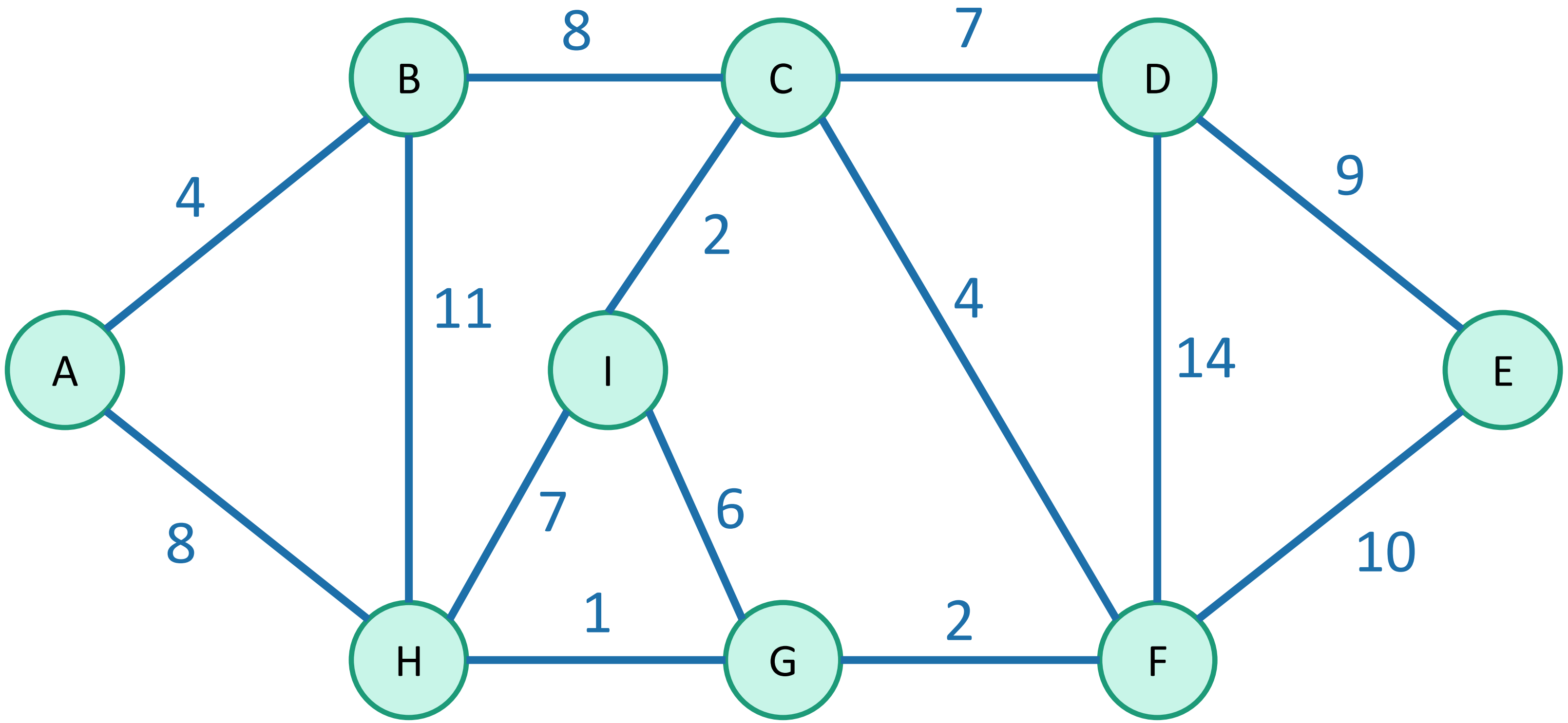
It has cost 37

A **tree** is a connected graph with no cycles!

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

Minimum Spanning Tree

Say we have an undirected weighted graph



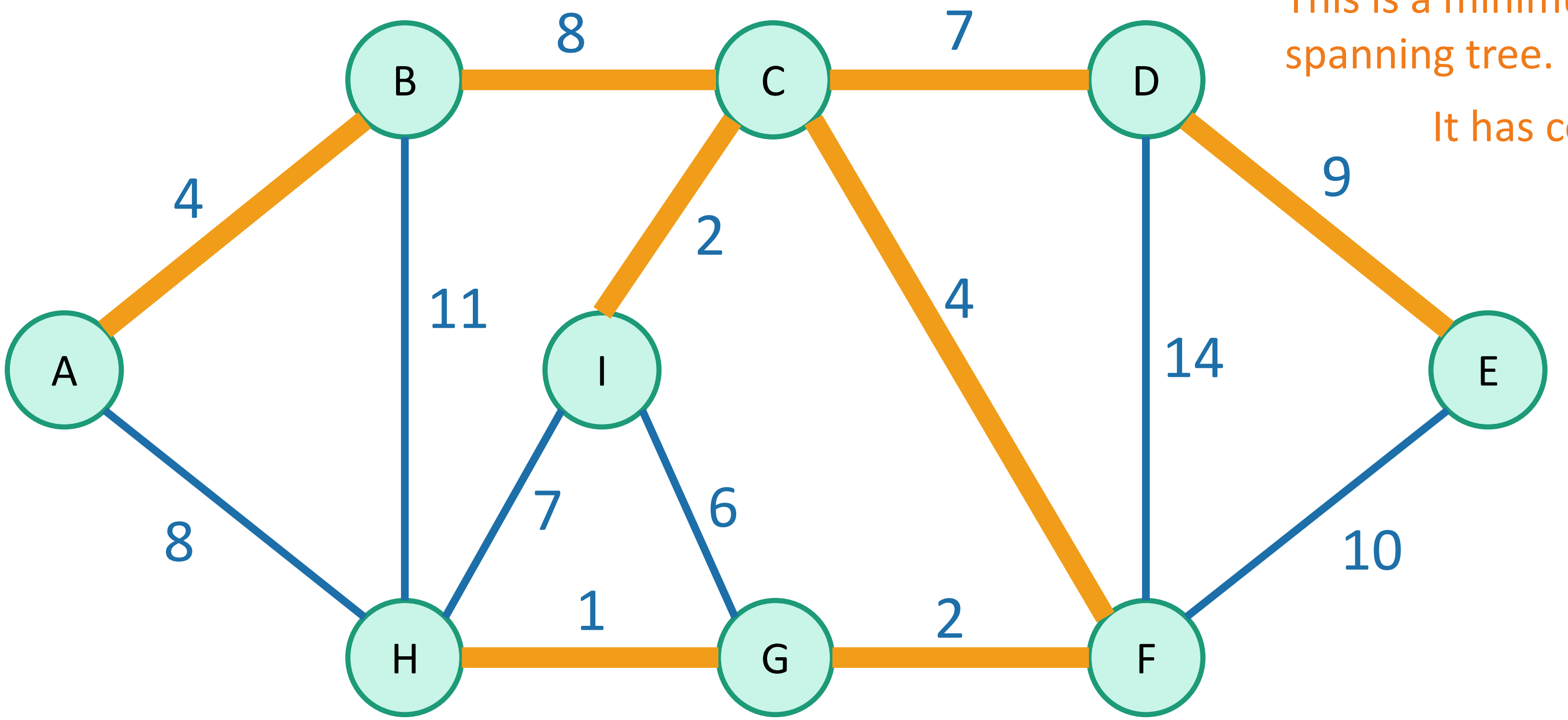
minimum

of minimal cost

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

Minimum Spanning Tree

Say we have an undirected weighted graph



This is a minimum spanning tree.

It has cost 37

minimum

of minimal cost

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

Why MSTs?

- Network design
  - Connecting cities with roads/electricity/telephone/…
- cluster analysis
  - eg, genetic distance
- image processing
  - eg, image segmentation
- Useful primitive
  - for other graph algs

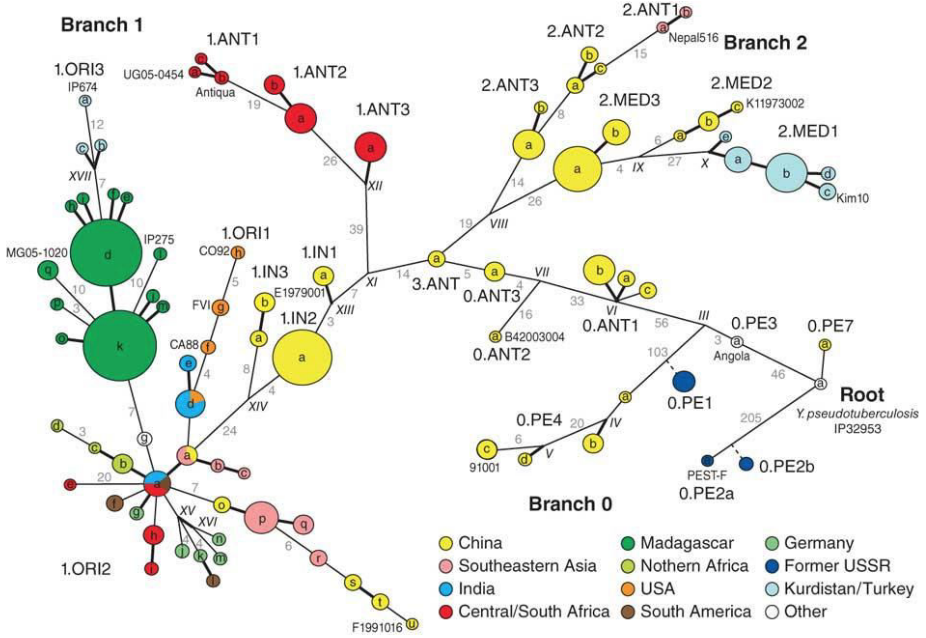
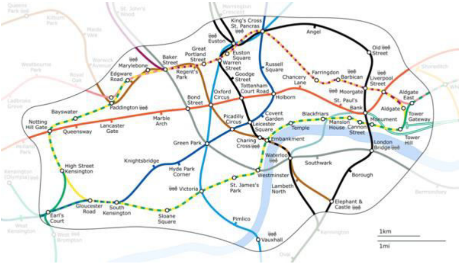


Figure 2: Fully parsimonious minimal spanning tree of 933 SNPs for 282 isolates of *Y. pestis* colored by location.

Morelli et al. Nature genetics 2010

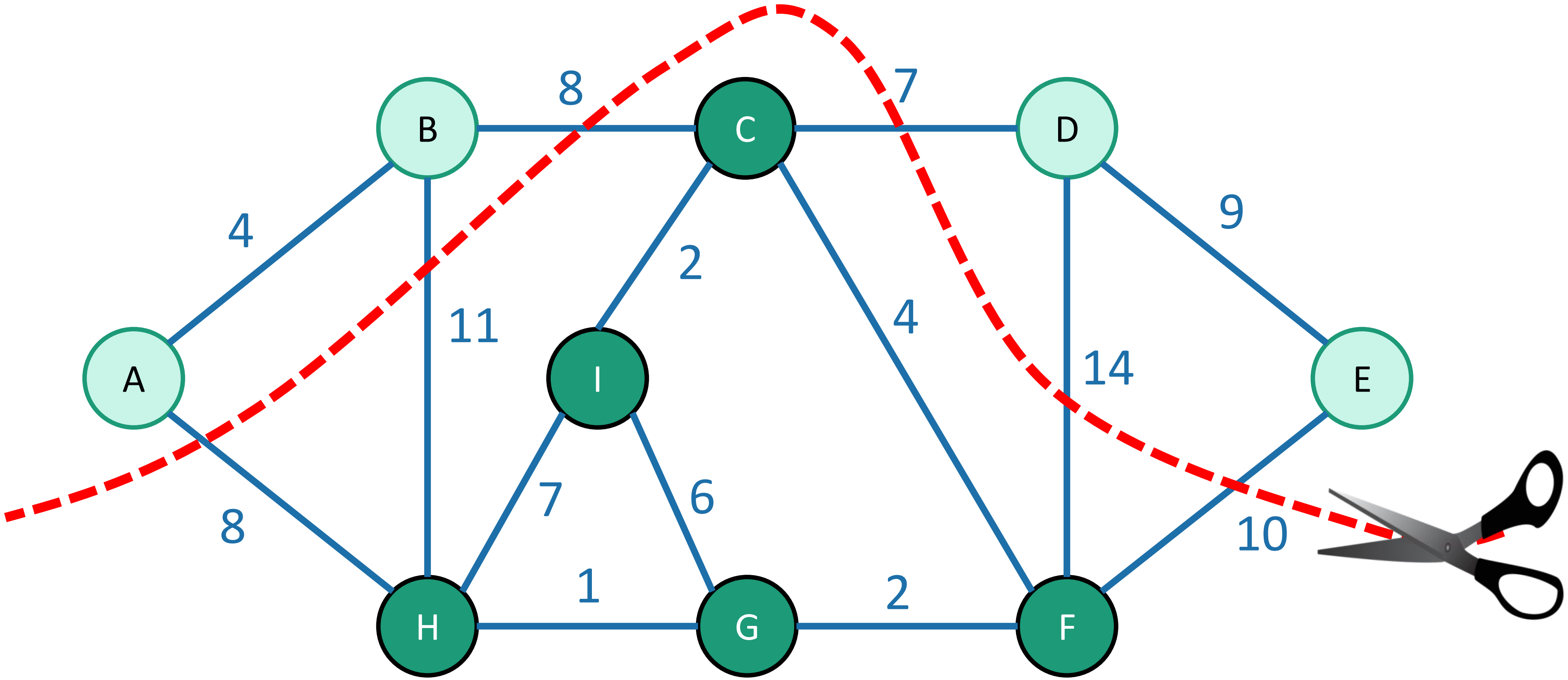


Brief aside

for a discussion of cuts in graphs!

Cuts in graphs

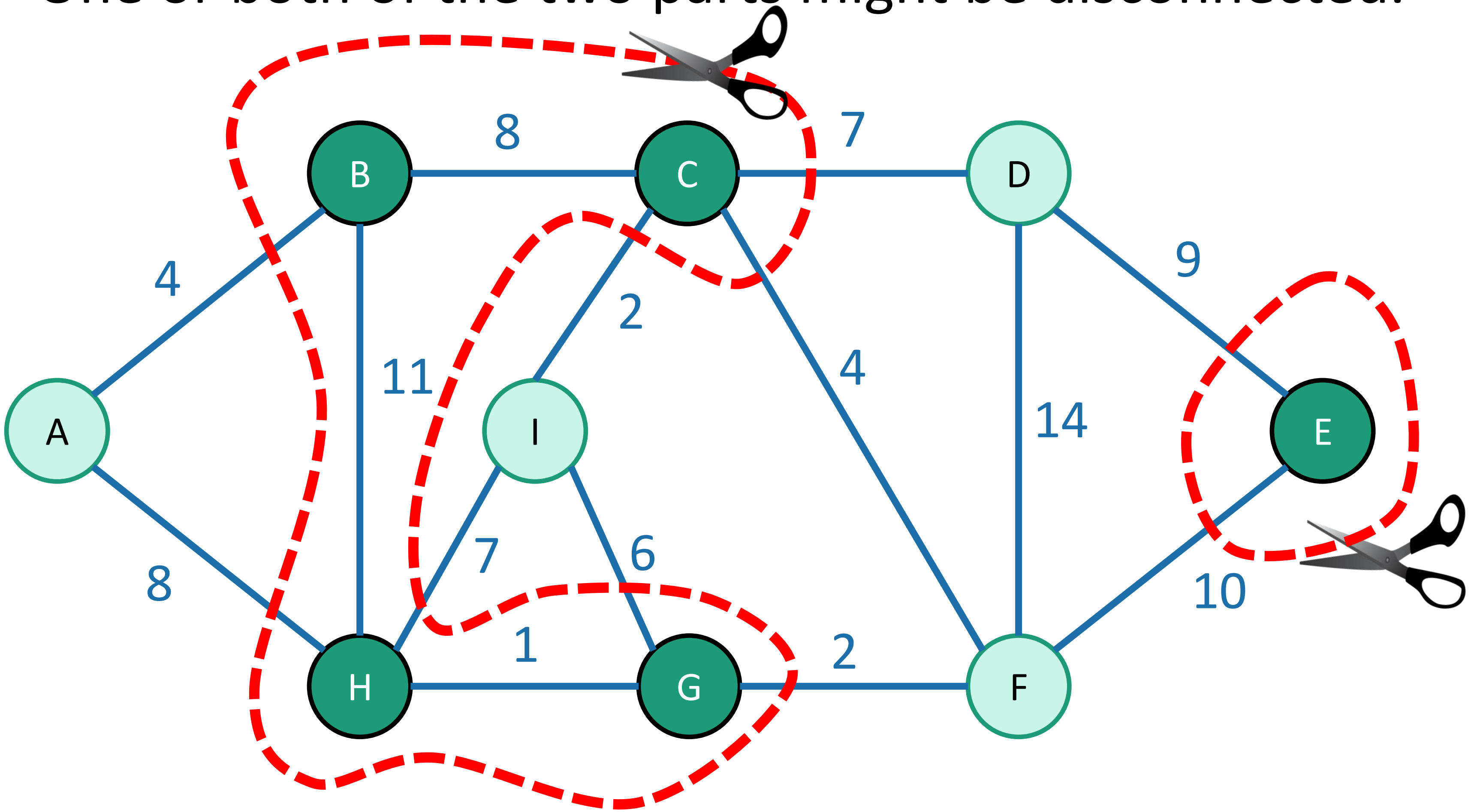
- A cut is a partition of the vertices into two parts:



This is the cut **"{A,B,D,E} and {C,I,H,G,F}"**

Cuts in graphs

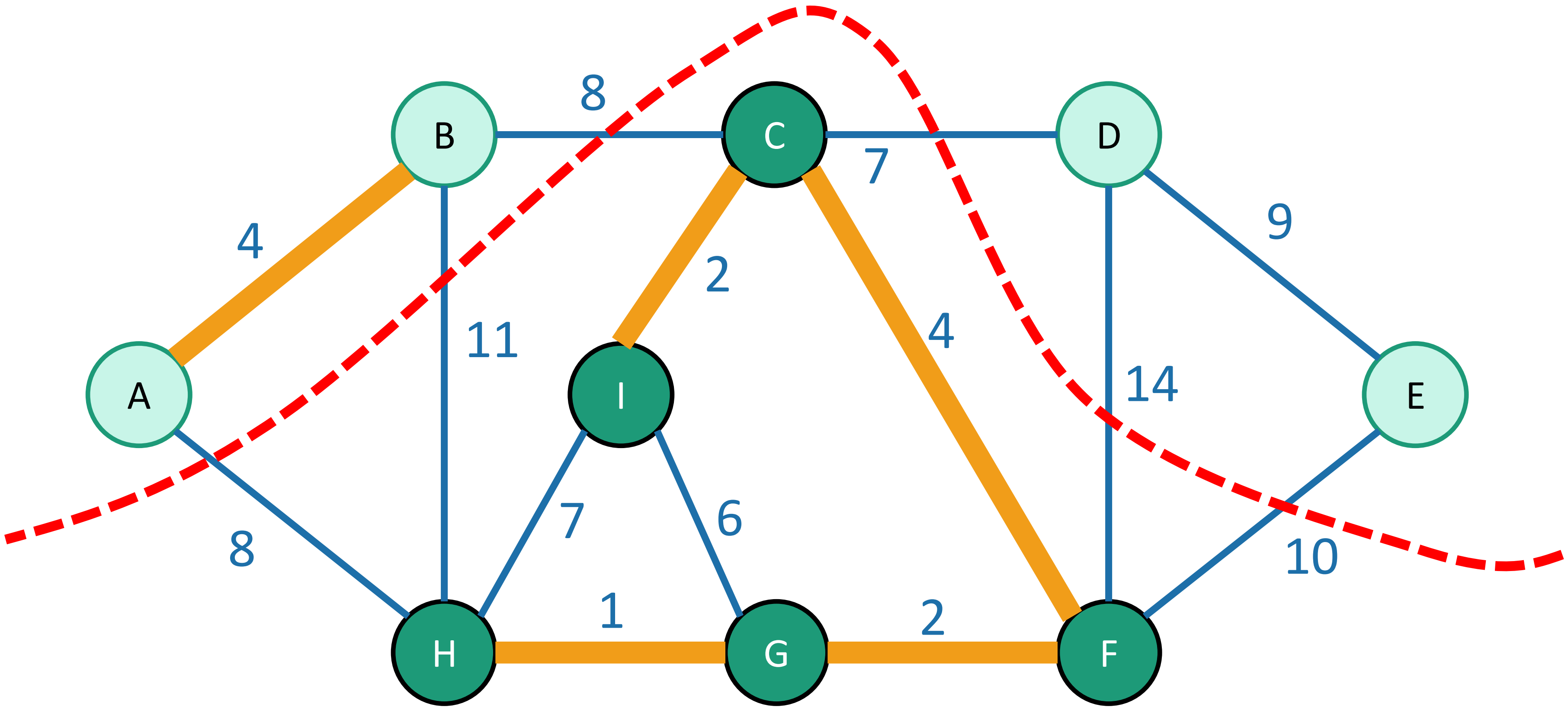
- One or both of the two parts might be disconnected.



This is the cut **"{B,C,E,G,H} and {A,D,I,F}"**

Let S be a set of edges in G

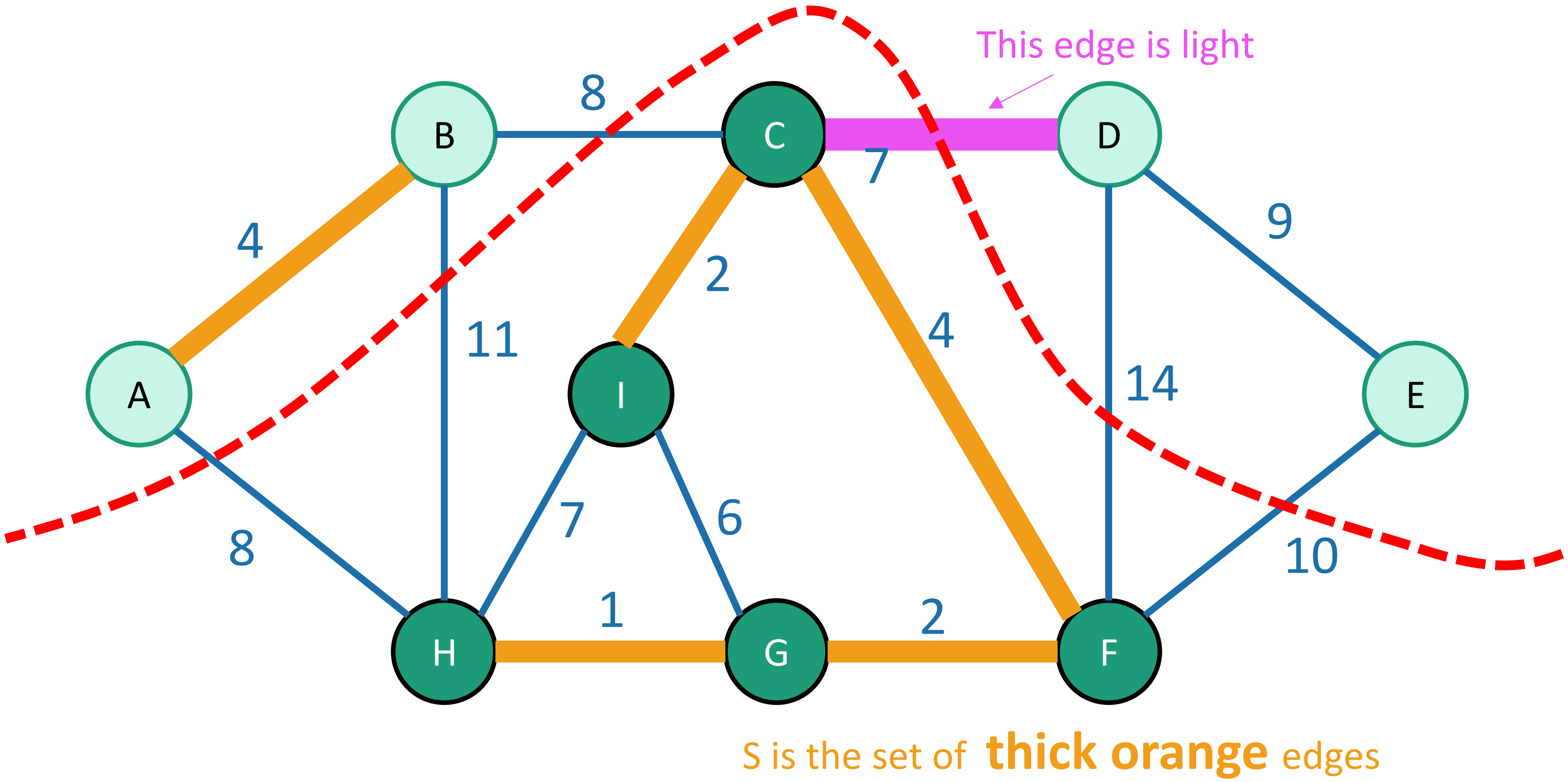
- We say a cut **respects** S if no edges in S cross the cut.
- An edge crossing a cut is called **light** if it has the smallest weight of any edge crossing the cut.



S is the set of **thick orange** edges

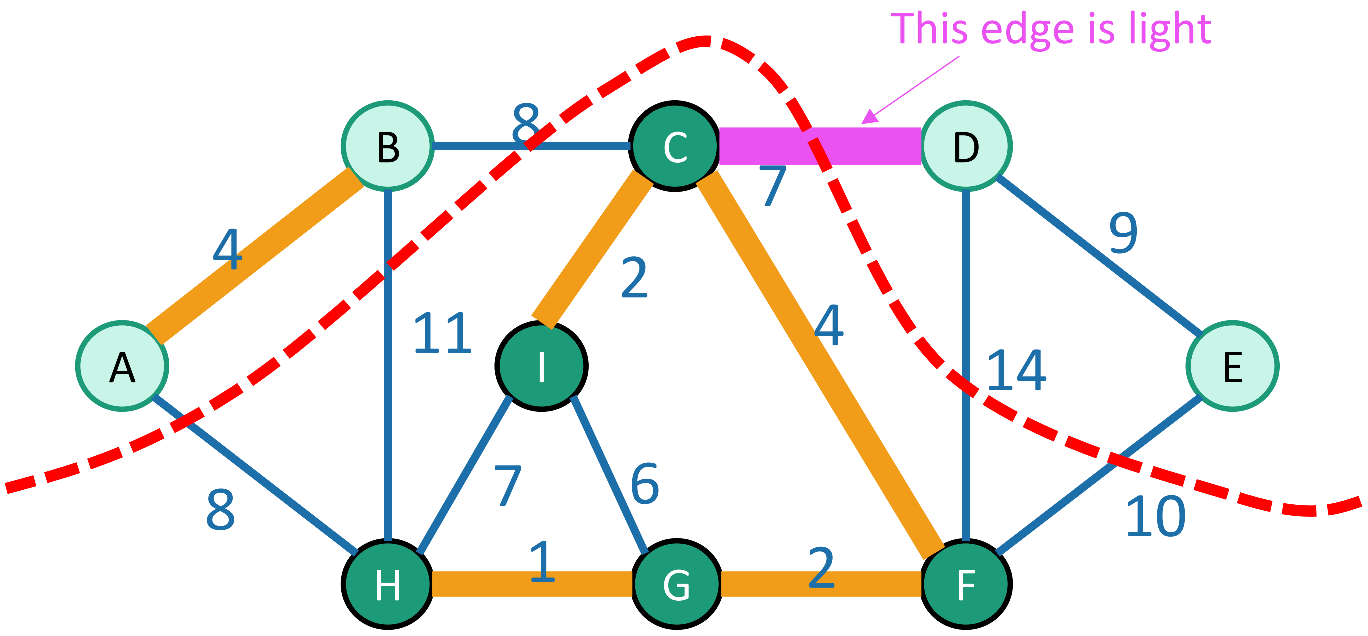
Let S be a set of edges in G

- We say a cut **respects** S if no edges in S cross the cut.
- An edge crossing a cut is called **light** if it has the smallest weight of any edge crossing the cut.



Lemma

- Let S be a set of edges, and consider a cut that respects S.
- Suppose there is an MST containing S.
- Let (u,v) be a light edge.
- Then there is an MST containing S ∪ {(u,v)}



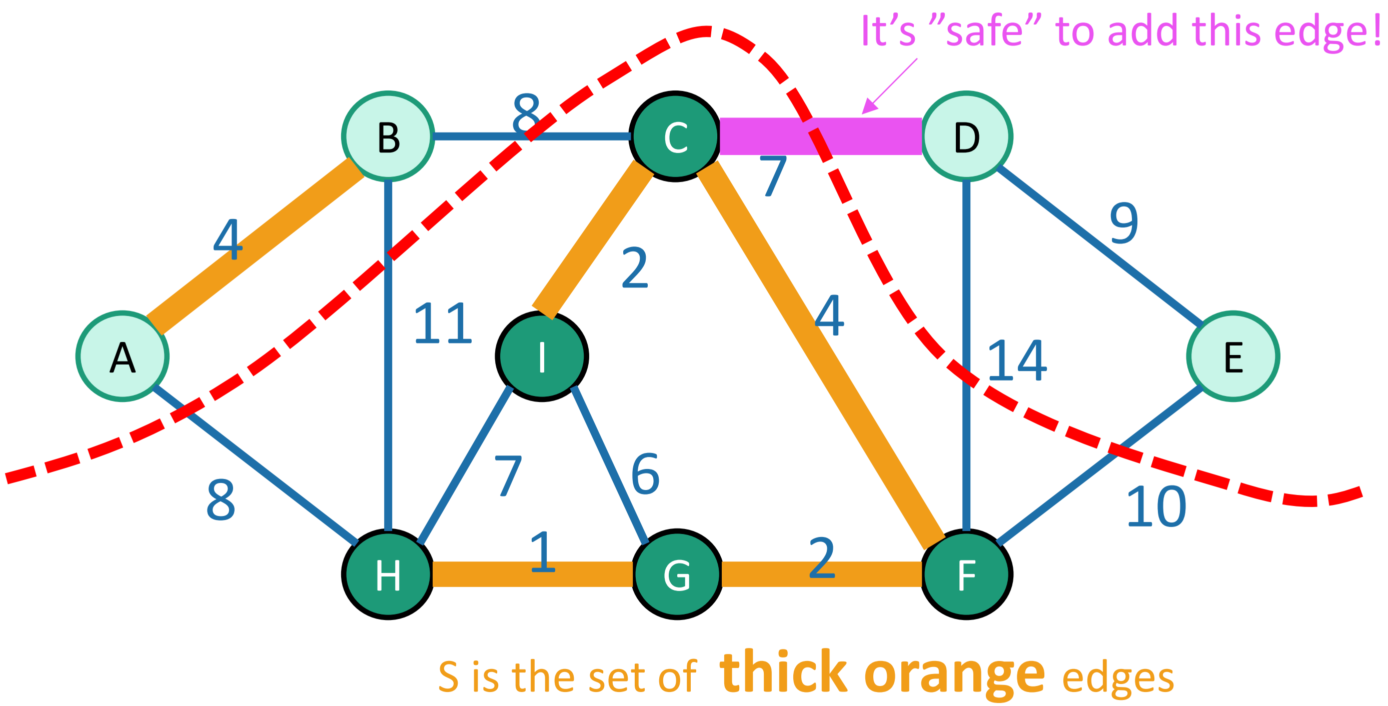
S is the set of **thick orange** edges

Lemma

- Let S be a set of edges, and consider a cut that respects S.
- Suppose there is an MST containing S.
- Let (u,v) be a light edge.
- Then there is an MST containing S ∪ {(u,v)}

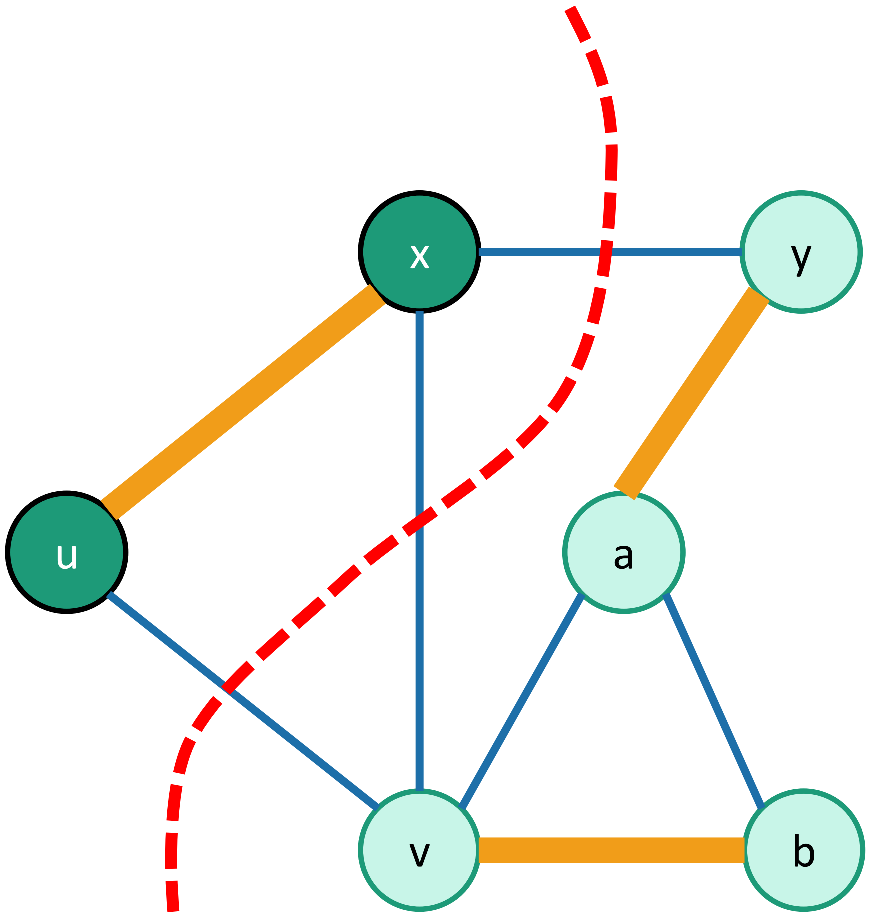
Aka:

If we haven't ruled out the possibility of success so far, then adding a light edge still won't rule it out.



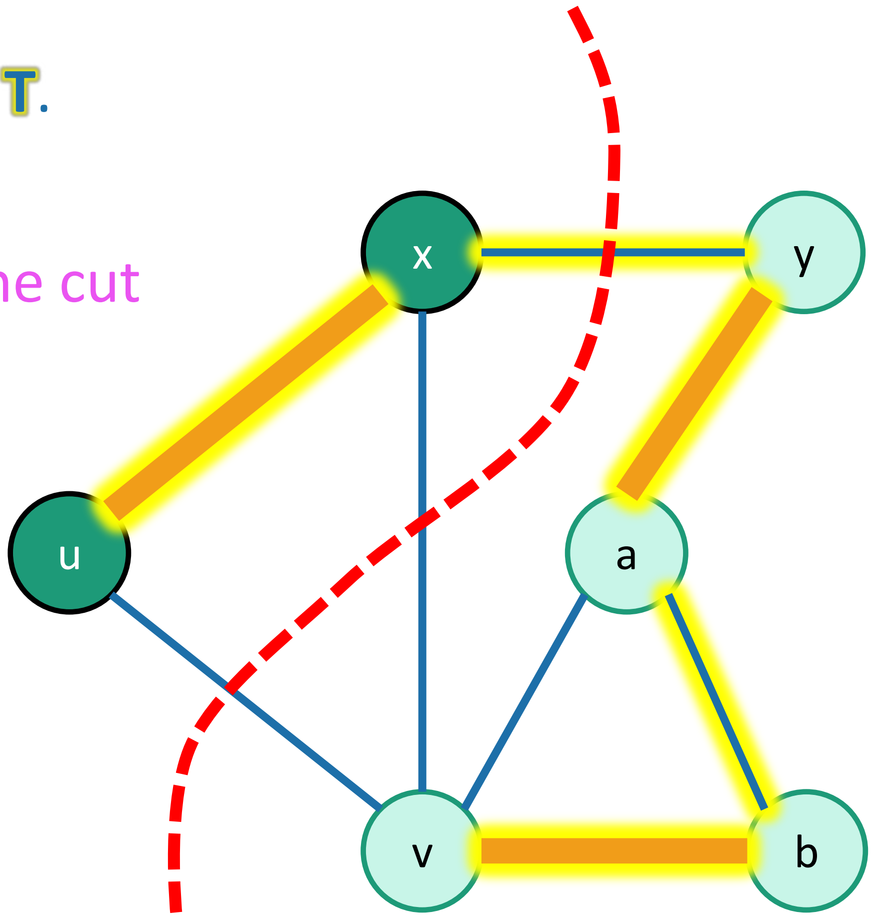
Proof of Lemma

- Assume that we have:
  - a **cut** that respects **S**



Proof of Lemma

- Assume that we have:
  - a **cut** that respects **S**
  - **S** is part of some **MST T**.
- Say that **(u,v)** is light.
  - lowest cost crossing the cut



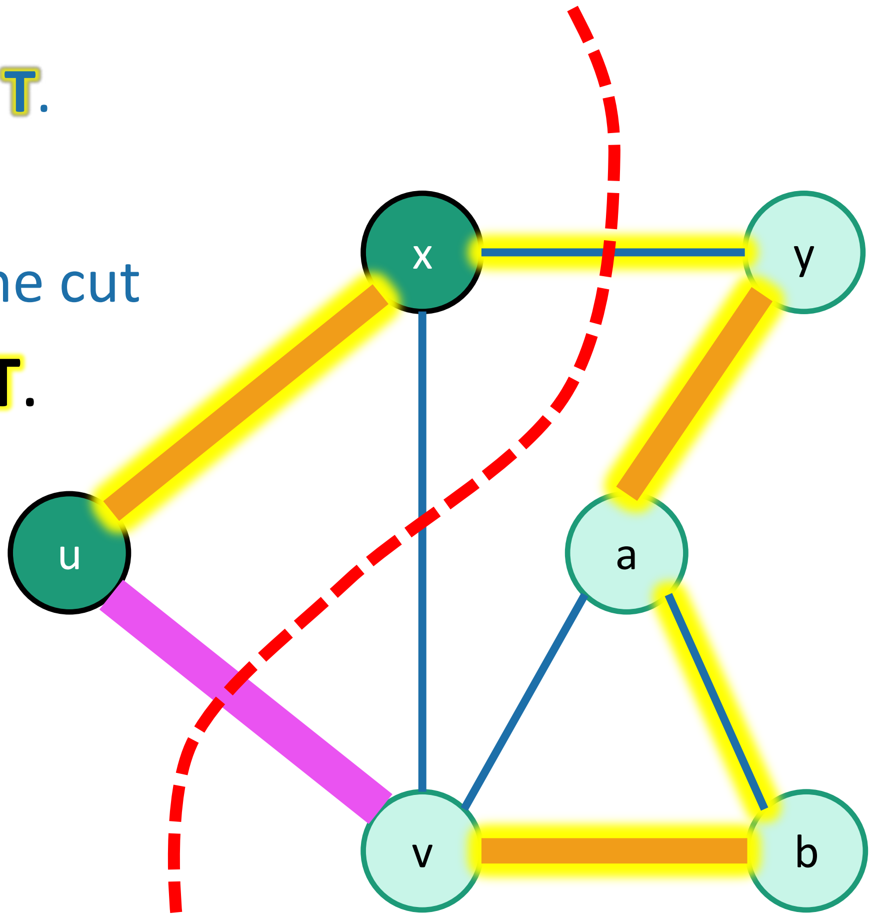
Proof of Lemma

Claim: Adding any additional edge to a spanning tree will create a cycle.

Proof: Both endpoints are already in the tree and connected to each other.

- Assume that we have:
  - a **cut** that respects **S**
  - **S** is part of some **MST T**.
- Say that **(u,v)** is light.
  - lowest cost crossing the cut
- But say **(u,v)** is not in **T**.
  - So adding **(u,v)** to **T** will make a cycle.

Otherwise we're done!

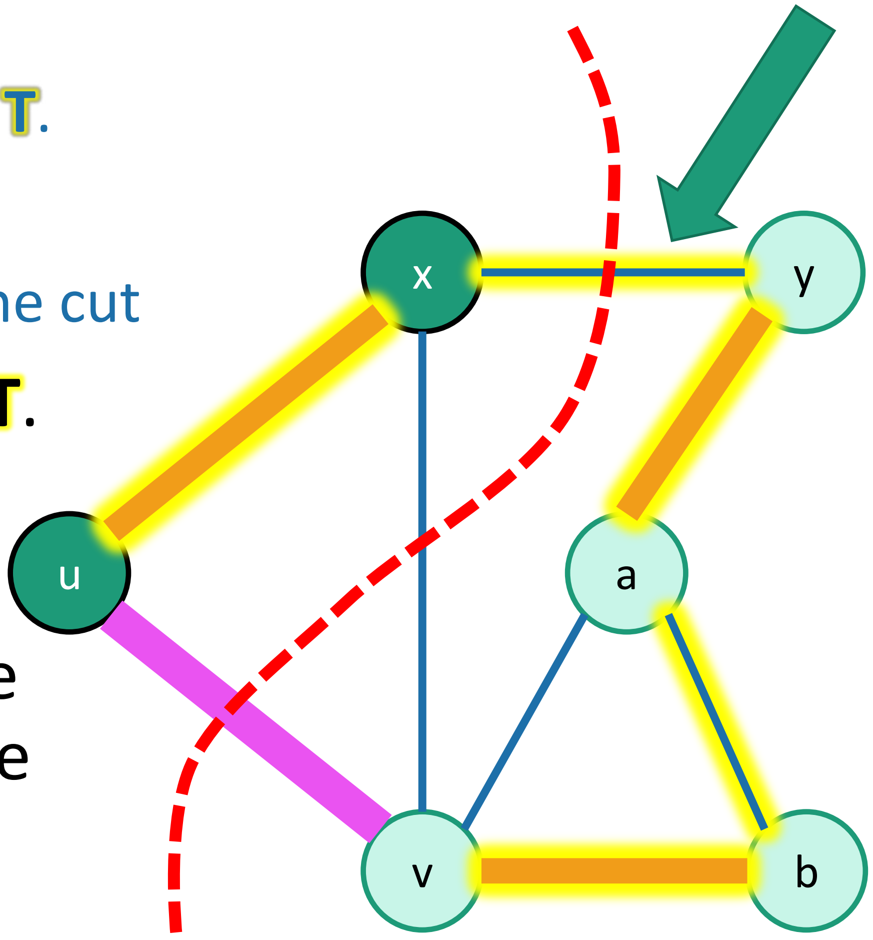


Proof of Lemma

Claim: Adding any additional edge to a spanning tree will create a cycle.

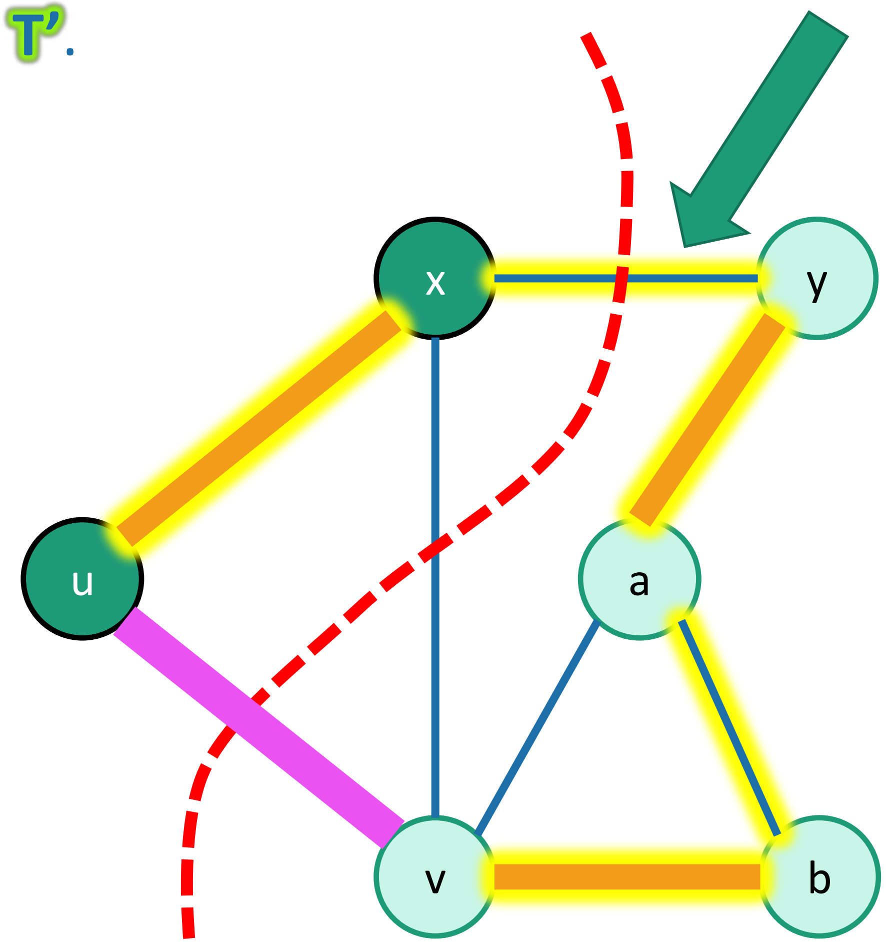
Proof: Both endpoints are already in the tree and connected to each other.

- Assume that we have:
  - a **cut** that respects **S**
  - **S** is part of some **MST T**.
- Say that **(u,v)** is light.
  - lowest cost crossing the cut
- But say **(u,v)** is not in **T**.
  - So adding **(u,v)** to **T** will make a cycle.
- So there is at least one other edge in this cycle crossing the cut.
  - call it (x,y)



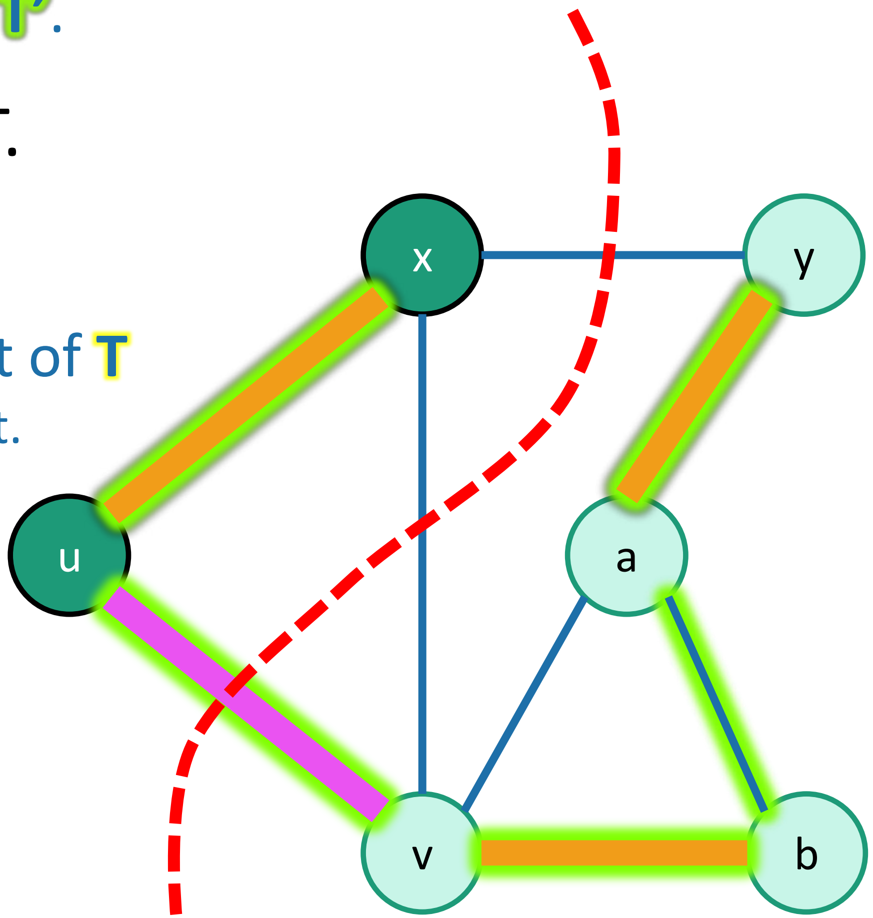
Proof of Lemma ctd.

- Consider swapping (u,v) for (x,y) in **T**.
  - Call the resulting tree **T'**.



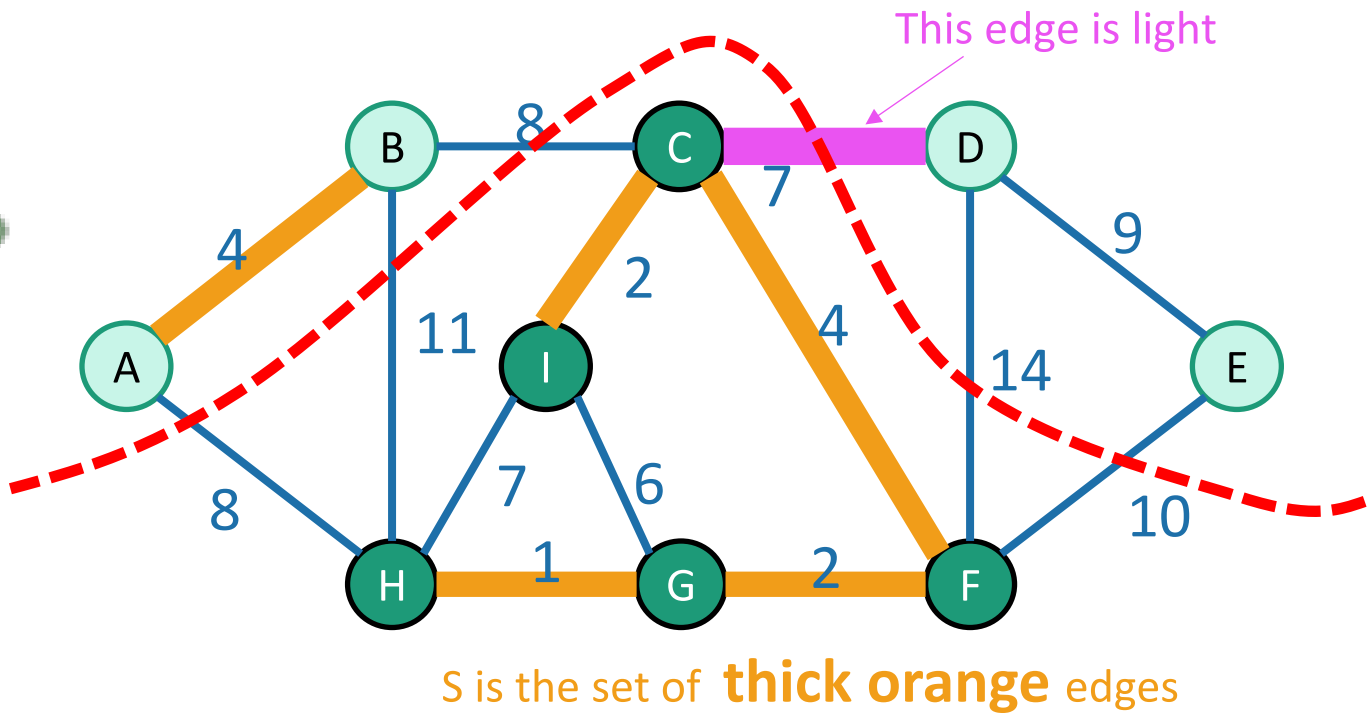
Proof of Lemma ctd.

- Consider swapping (u,v) for (x,y) in T.
  - Call the resulting tree T'.
- **Claim**: T' is still an MST.
  - It is still a tree:
    - we deleted (x,y)
  - It has cost at most that of T
    - because (u,v) was light.
  - T had minimal cost.
  - So T' does too.
- So T' is an MST containing (u,v).
  - This is what we wanted.



Lemma

- Let S be a set of edges, and consider a cut that respects S.
- Suppose there is an MST containing S.
- Let (u,v) be a light edge.
- Then there is an MST containing S ∪ {(u,v)}



End aside

Back to MSTs!

Back to MSTs

- How do we find one?
- Today we'll see **two greedy algorithms**.

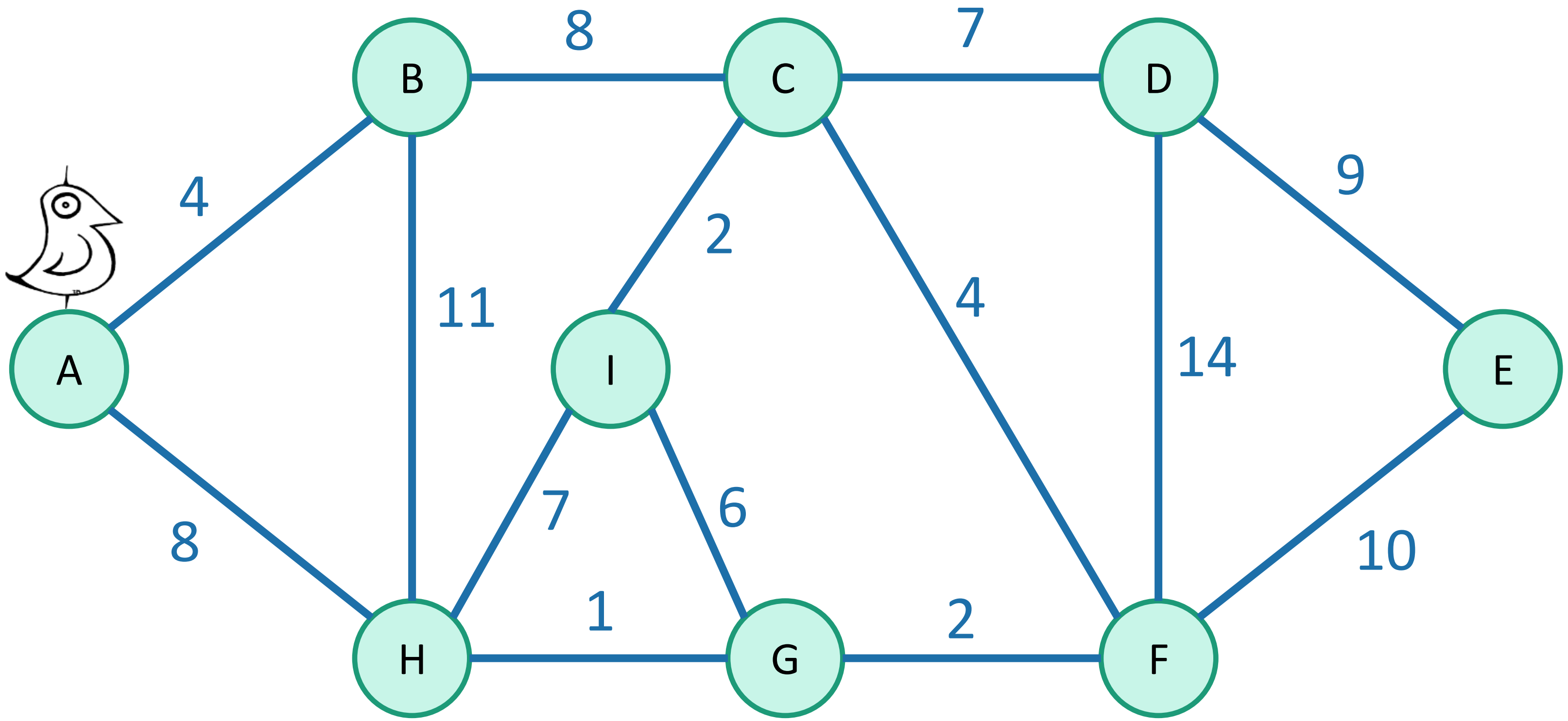
- The strategy:
  - Make a **series of choices,** adding edges to the tree.
  - Show that each edge we add is **safe to add**:
    - we do not rule out the possibility of success
    - we will choose **light edges** crossing **cuts** and **use the Lemma**.
  - **Keep going** until we have an MST.

Prim’s Algorithm

Minimum Spanning Tree

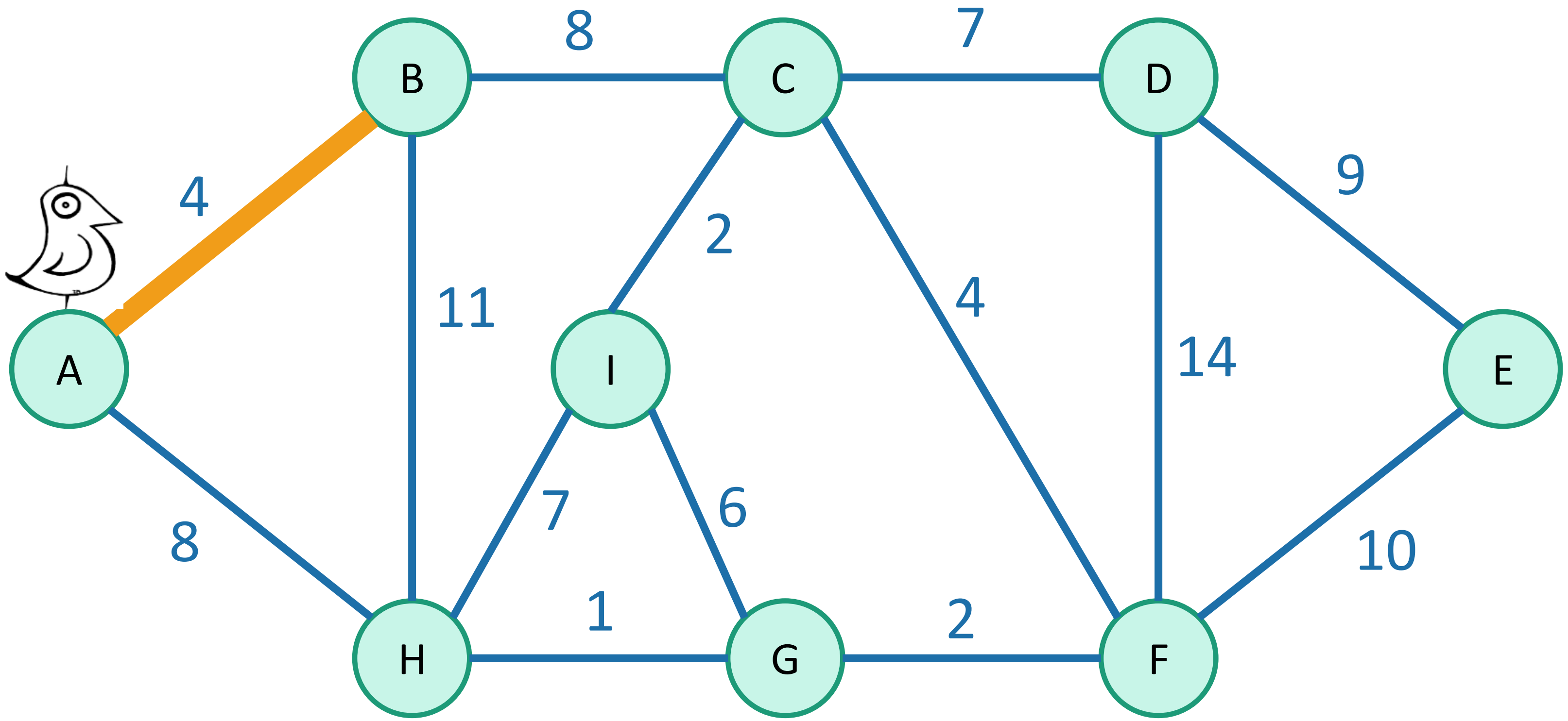
Idea 1

Start growing a tree, greedily add the shortest edge we can to grow the tree.



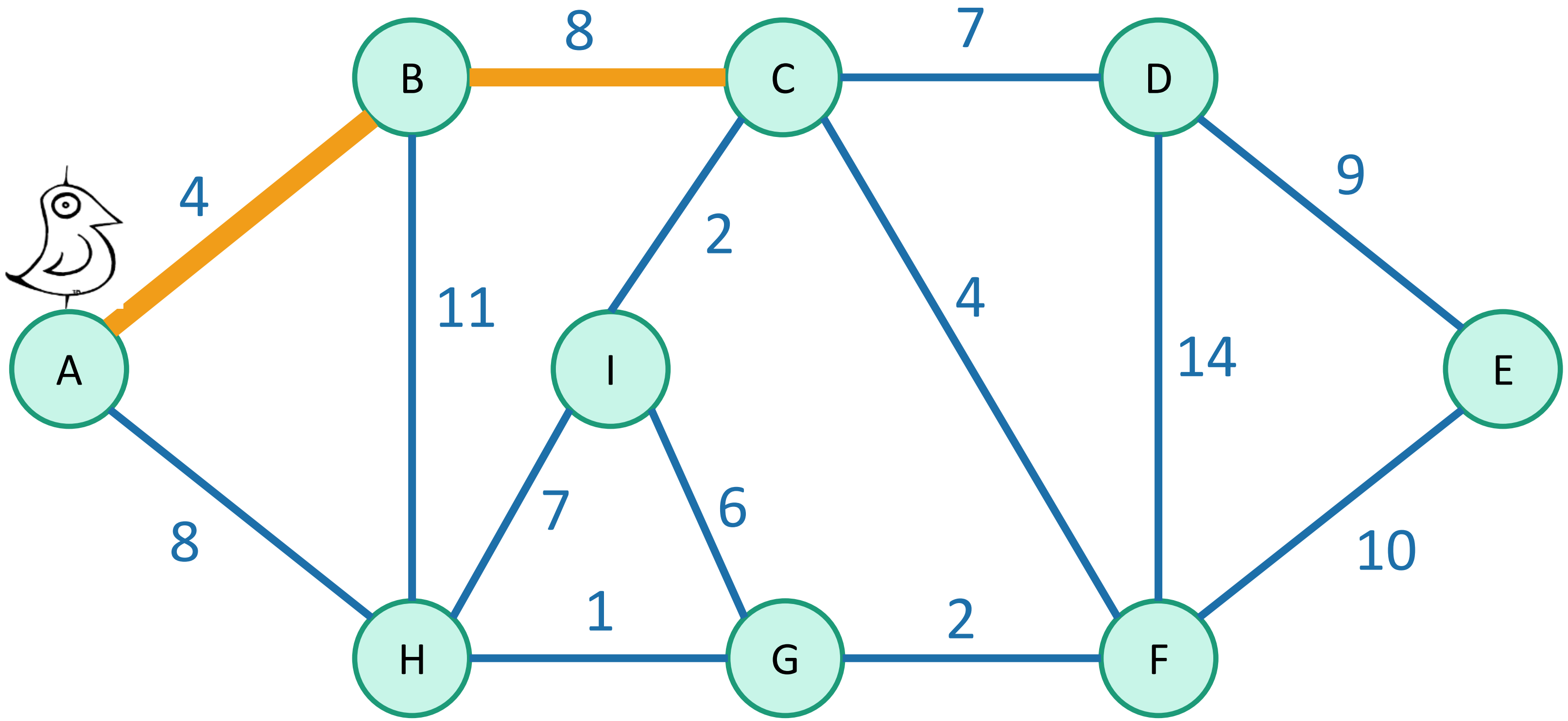
Idea 1

Start growing a tree, greedily add the shortest edge we can to grow the tree.



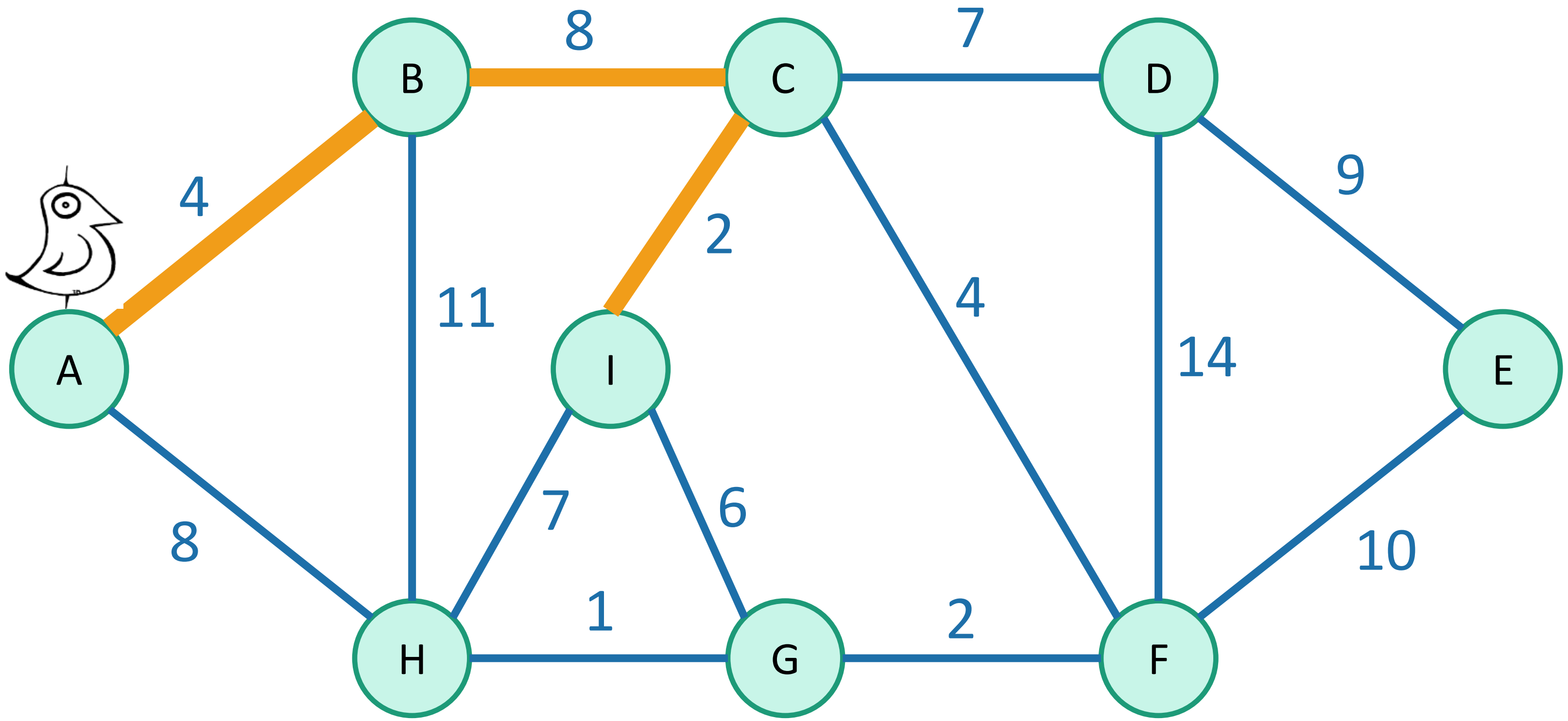
Idea 1

Start growing a tree, greedily add the shortest edge we can to grow the tree.



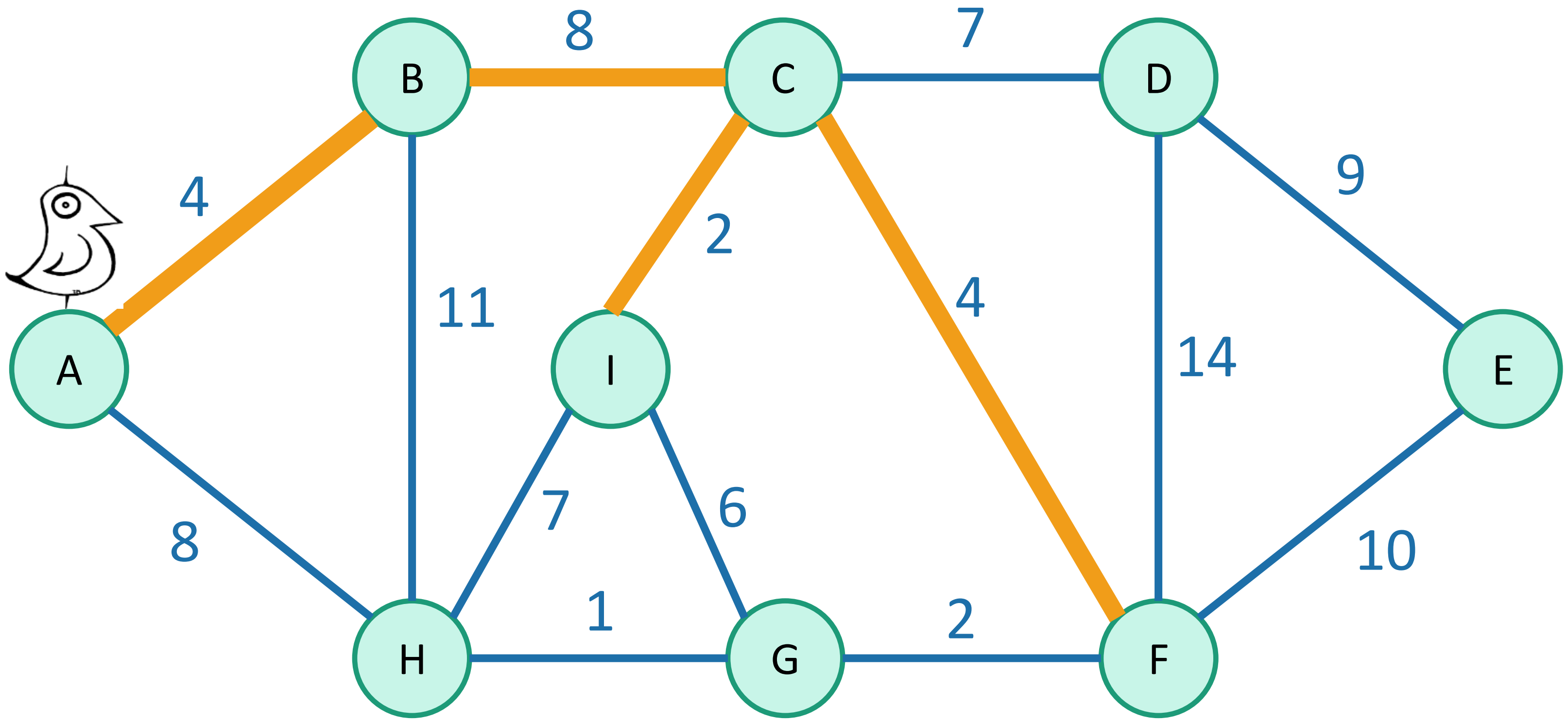
Idea 1

Start growing a tree, greedily add the shortest edge we can to grow the tree.



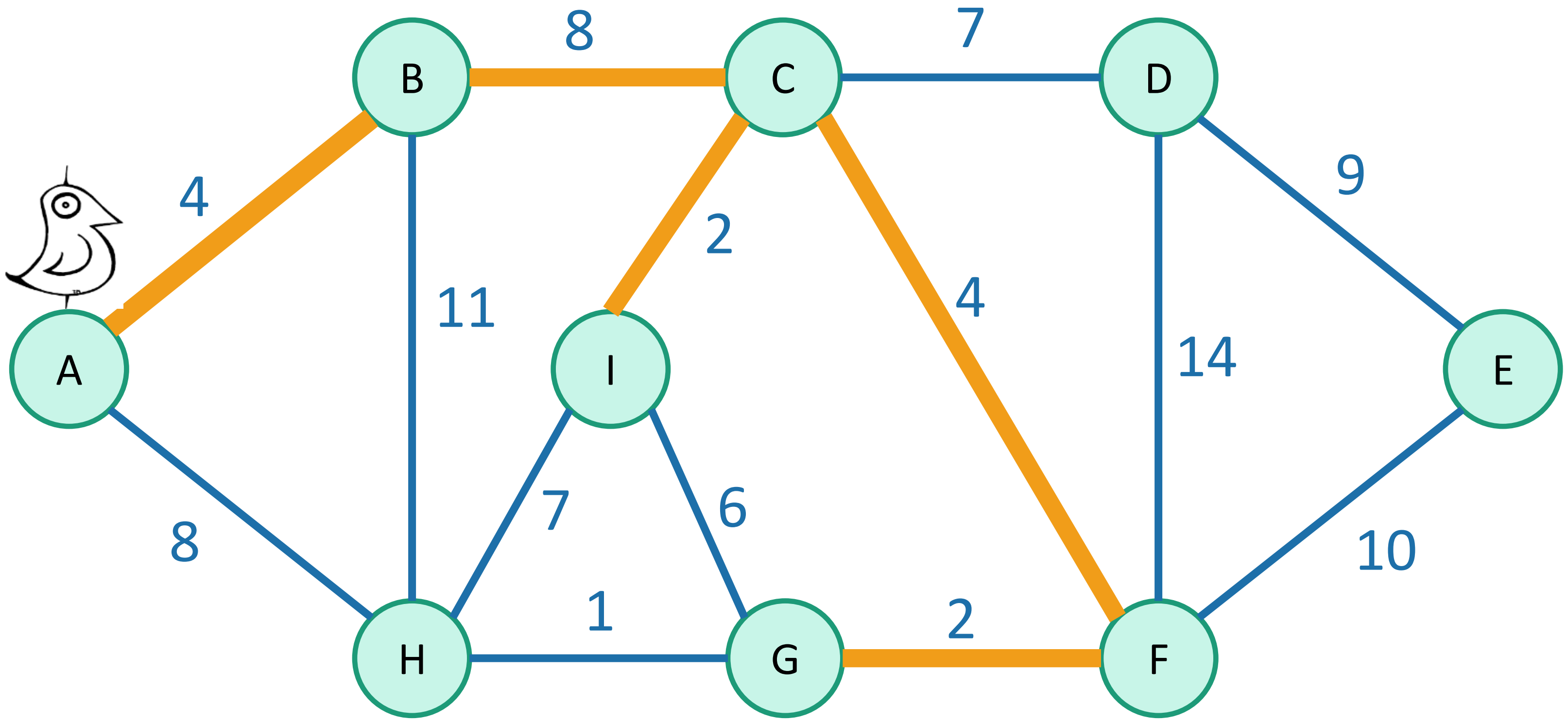
Idea 1

Start growing a tree, greedily add the shortest edge we can to grow the tree.



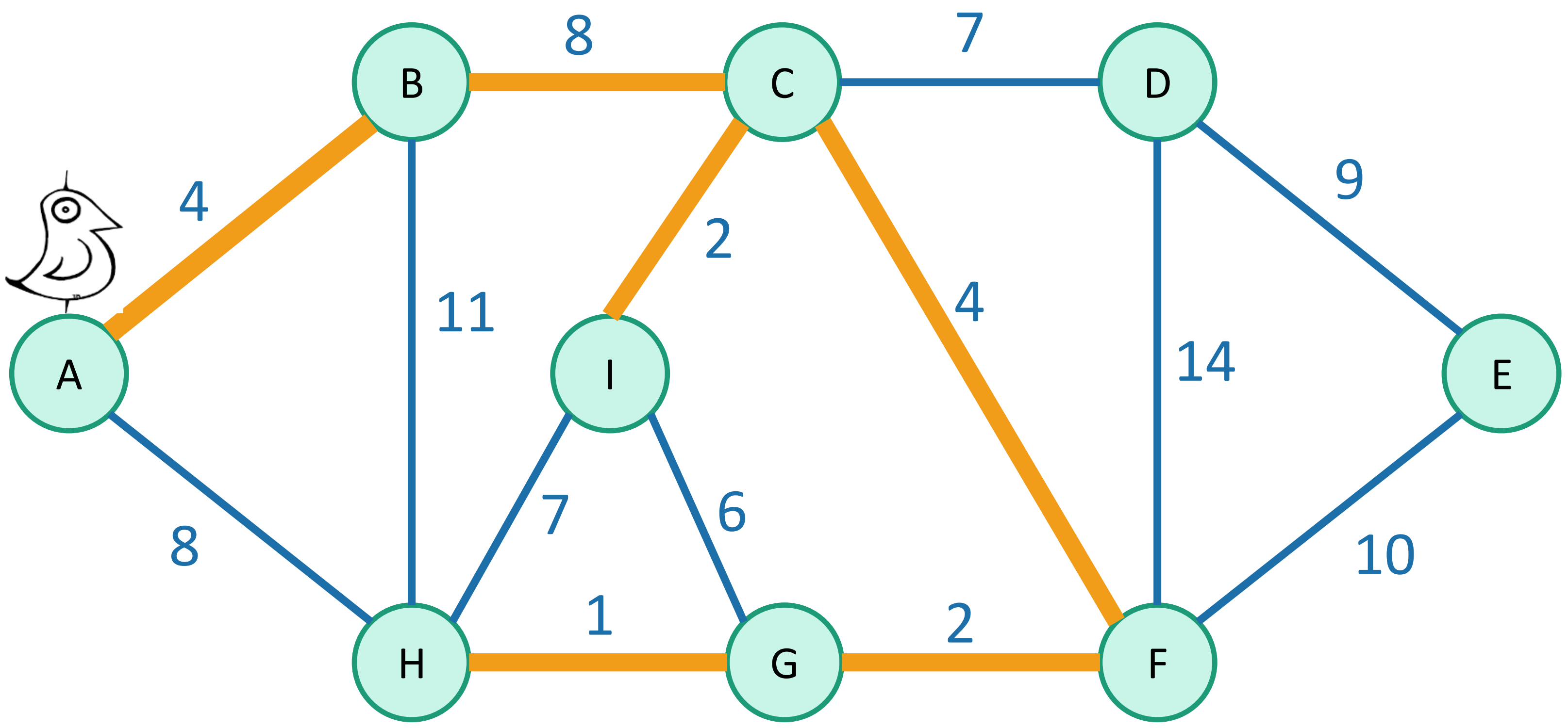
Idea 1

Start growing a tree, greedily add the shortest edge we can to grow the tree.



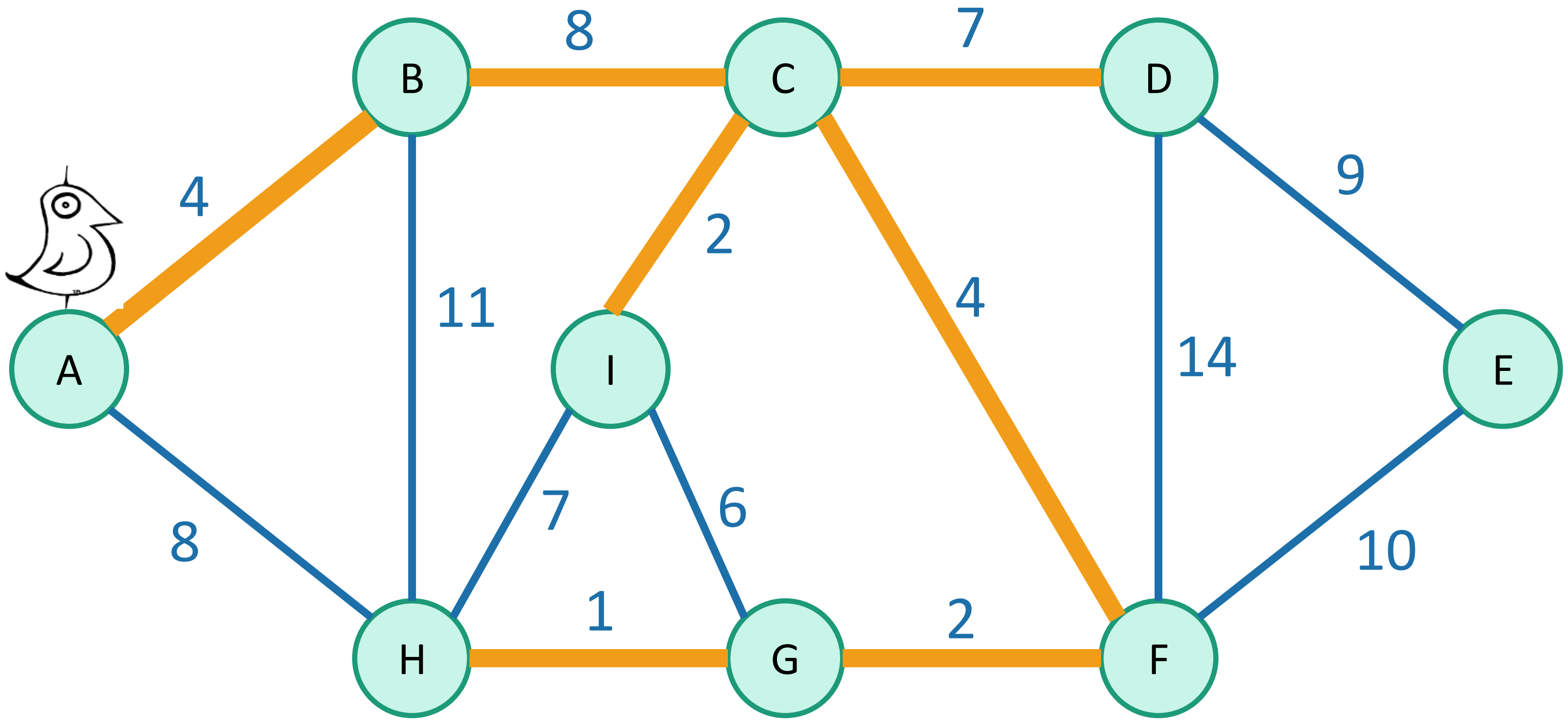
Idea 1

Start growing a tree, greedily add the shortest edge we can to grow the tree.



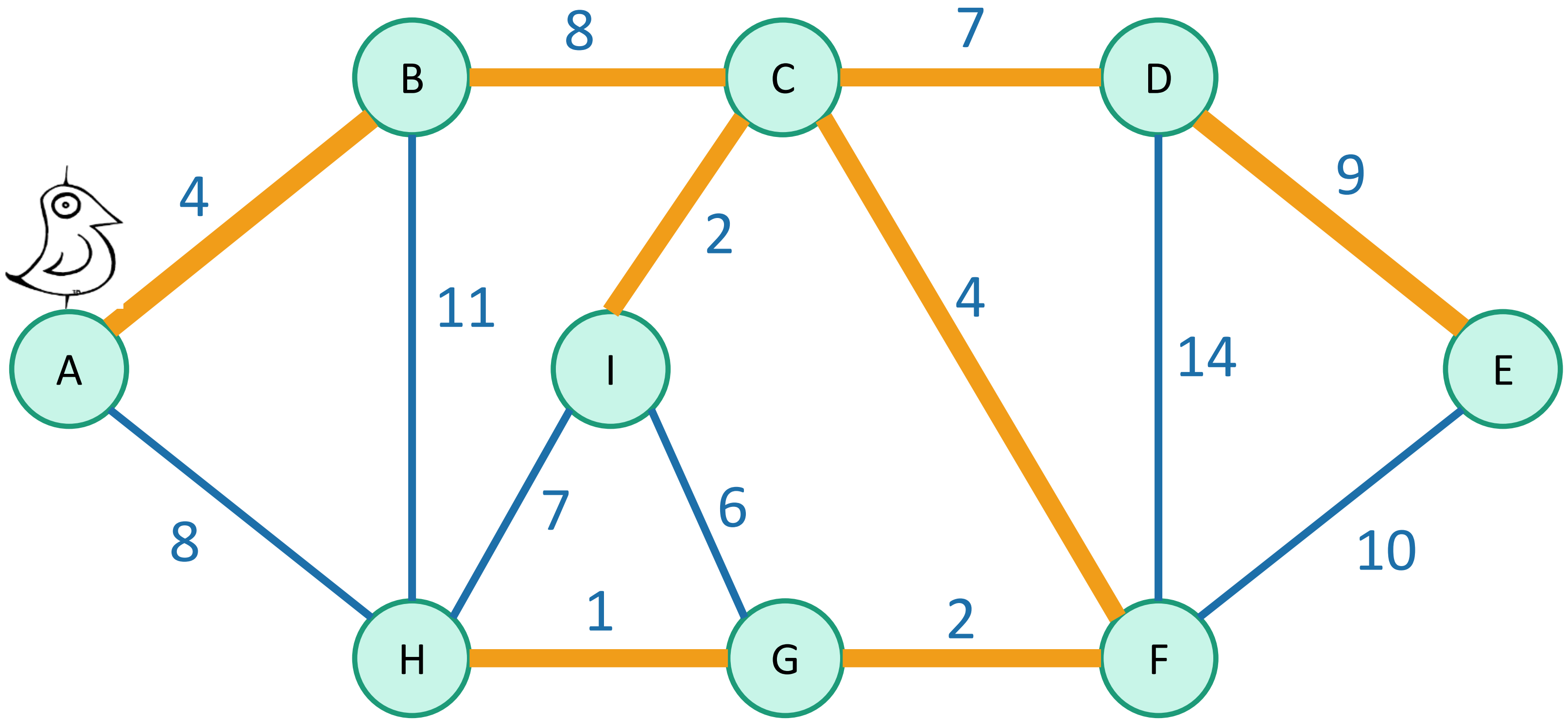
Idea 1

Start growing a tree, greedily add the shortest edge we can to grow the tree.



Idea 1

Start growing a tree, greedily add the shortest edge we can to grow the tree.



We've discovered

Prim's algorithm!

- slowPrim(G = (V,E), starting vertex s):
  - Let (s,u) be the lightest edge coming out of s.
  - MST = { (s,u) }
  - verticesVisited = { s, u }
  - **while** |verticesVisited| < |V|:
    - find the lightest edge (x,v) in E so that:
      - x is in verticesVisited
      - v is not in verticesVisited
    - add (x,v) to MST
    - add v to verticesVisited
  - **return** MST

n iterations of this while loop.

Maybe take time m to go through all the edges and find the lightest.

<u>Naively, the running time is O(nm):</u>

- For each of n-1 iterations of the while loop:
  - Maybe go through all the edges.

Two questions

1. Does it work?
   - That is, does it actually return a MST?

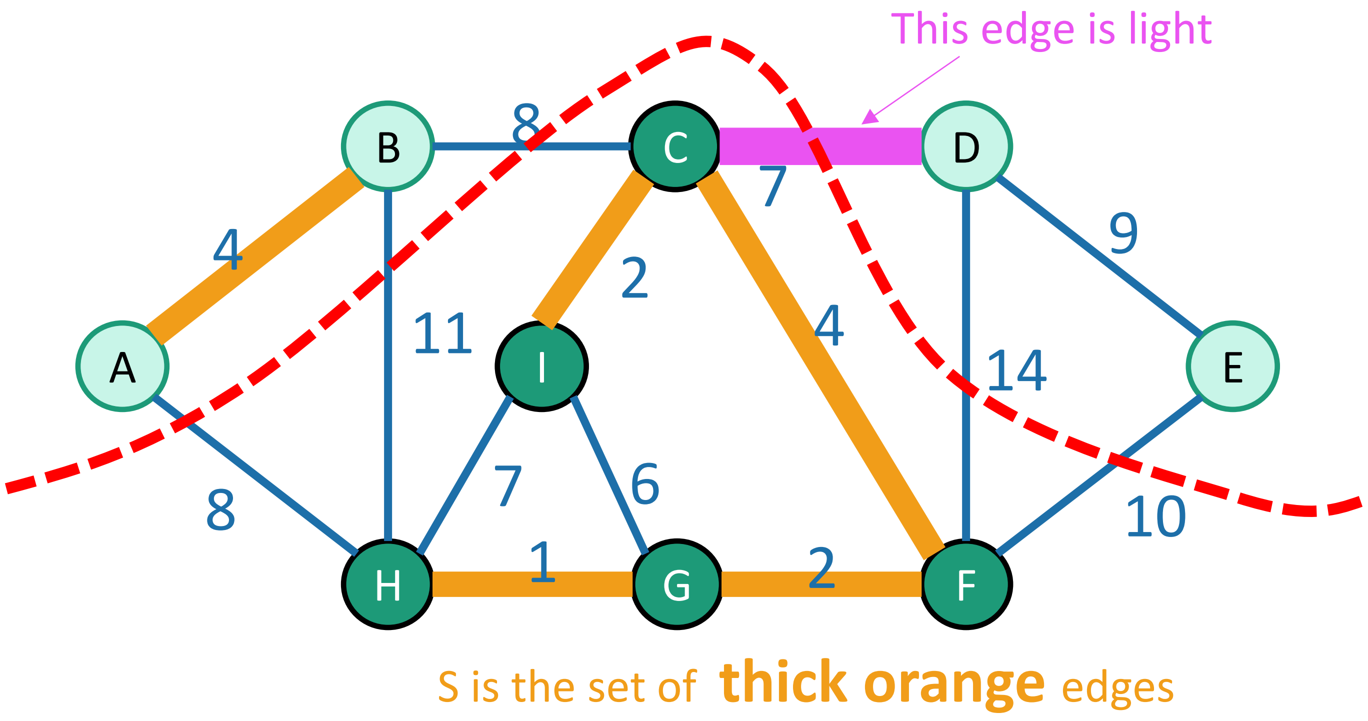
2. How do we actually implement this?
   - the pseudocode above says “slowPrim”…

Does it work?

- We need to show that our greedy choices **don't rule out success.**
- That is, at every step:
  - There exists an MST that contains all of the edges we have added so far.
- Now it is time to use our lemma!

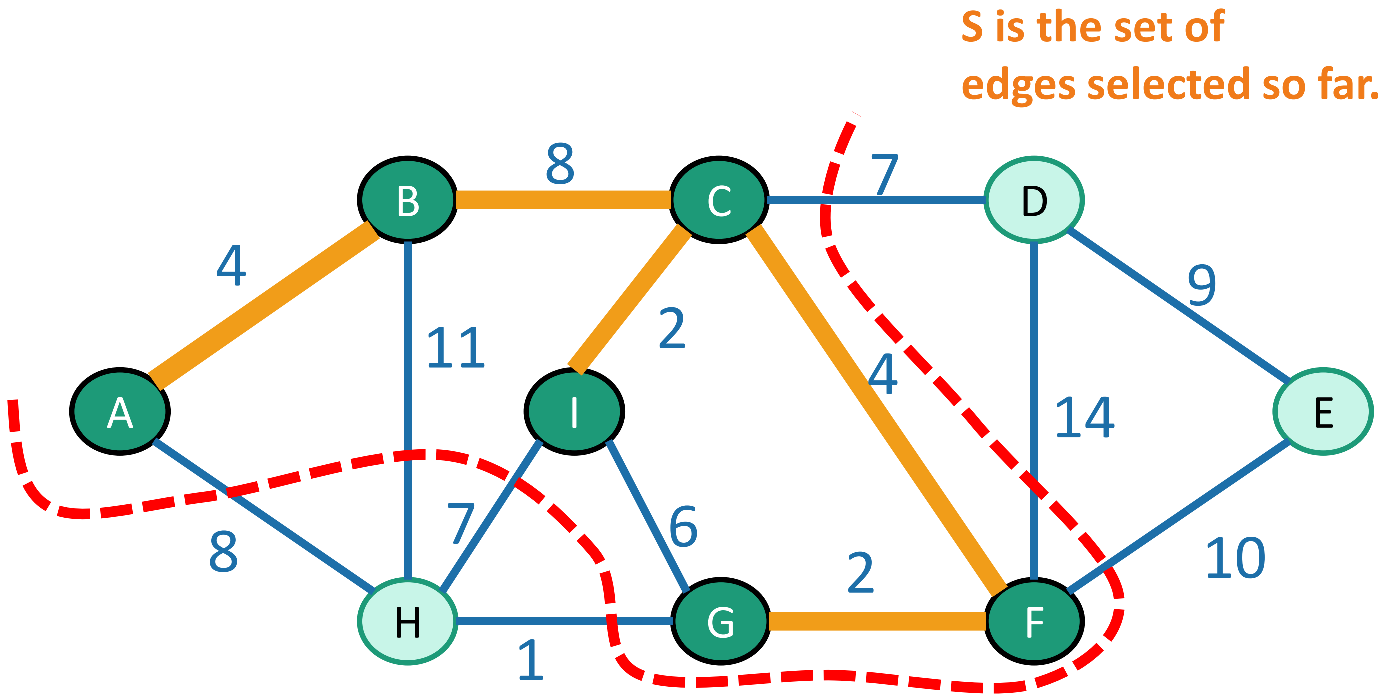
Lemma

- Let S be a set of edges, and consider a cut that respects S.
- Suppose there is an MST containing S.
- Let (u,v) be a light edge.
- Then there is an MST containing S ∪ {(u,v)}



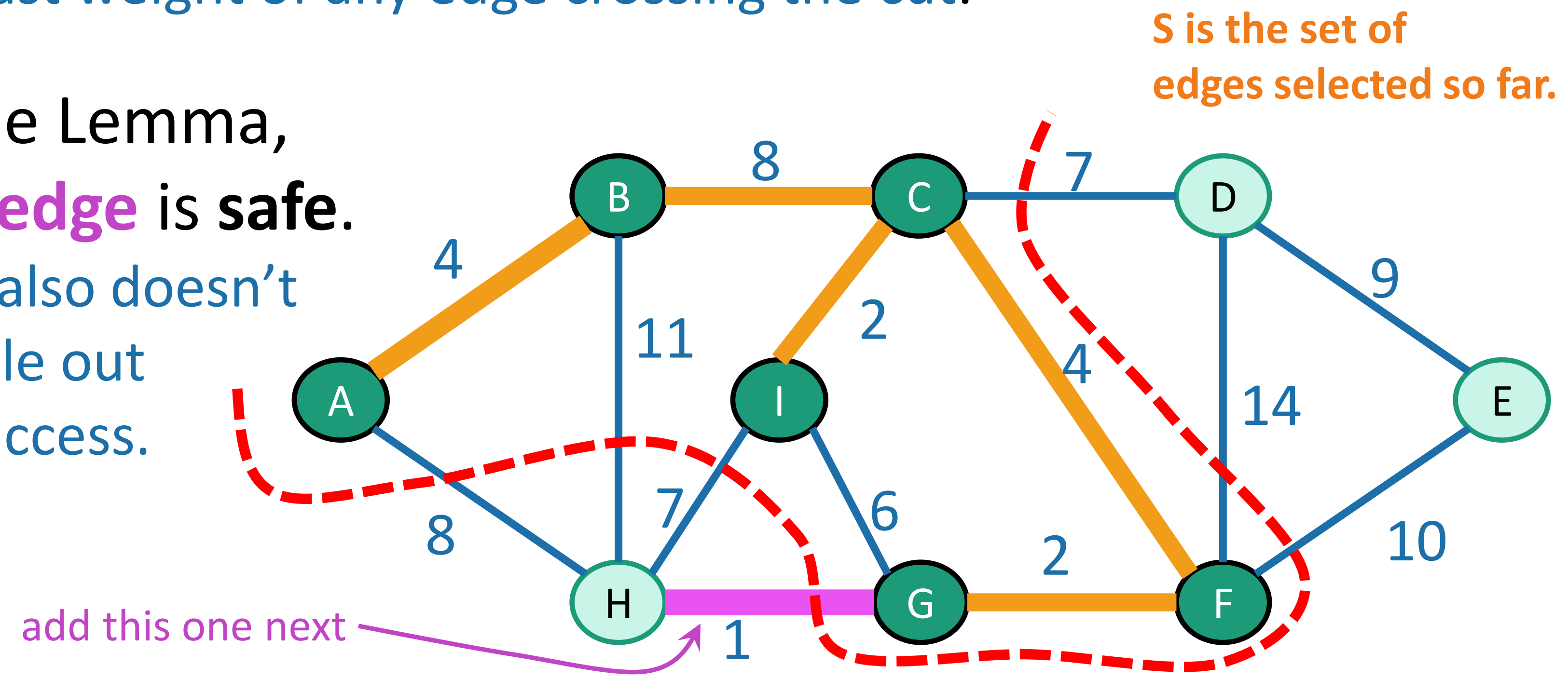
Partway through Prim

- Assume that our choices **S** so far are **safe.**
  - they don't rule out success
- Consider the cut {**visited**, **unvisited**}
  - This cut respects S.



Partway through Prim

- Assume that our choices **S** so far are **safe.**
  - they don't rule out success
- Consider the cut {**visited**, **unvisited**}
  - S respects this cut.
- The edge we add next is a **light edge**.
  - Least weight of any edge crossing the cut.
- By the Lemma, **that edge** is **safe**.
  - it also doesn't rule out success.



Hooray!

- Our greedy choices **don't rule out success**.
- This is enough (along with an argument by induction) to guarantee correctness of Prim's algorithm.

Formally(ish)

- Inductive hypothesis:
  - After adding the t'th edge, there exists an MST with the edges added so far.
- Base case:
  - After adding the 0'th edge, there exists an MST with the edges added so far. **YEP.**
- Inductive step:
  - If the inductive hypothesis holds for t (aka, the choices so far are safe), then it holds for t+1 (aka, the next edge we add is safe).
  - **That's what we just showed.**
- Conclusion:
  - After adding the n-1'st edge, there exists an MST with the edges added so far.
  - At this point we have a spanning tree, so it better be minimal.

Two questions

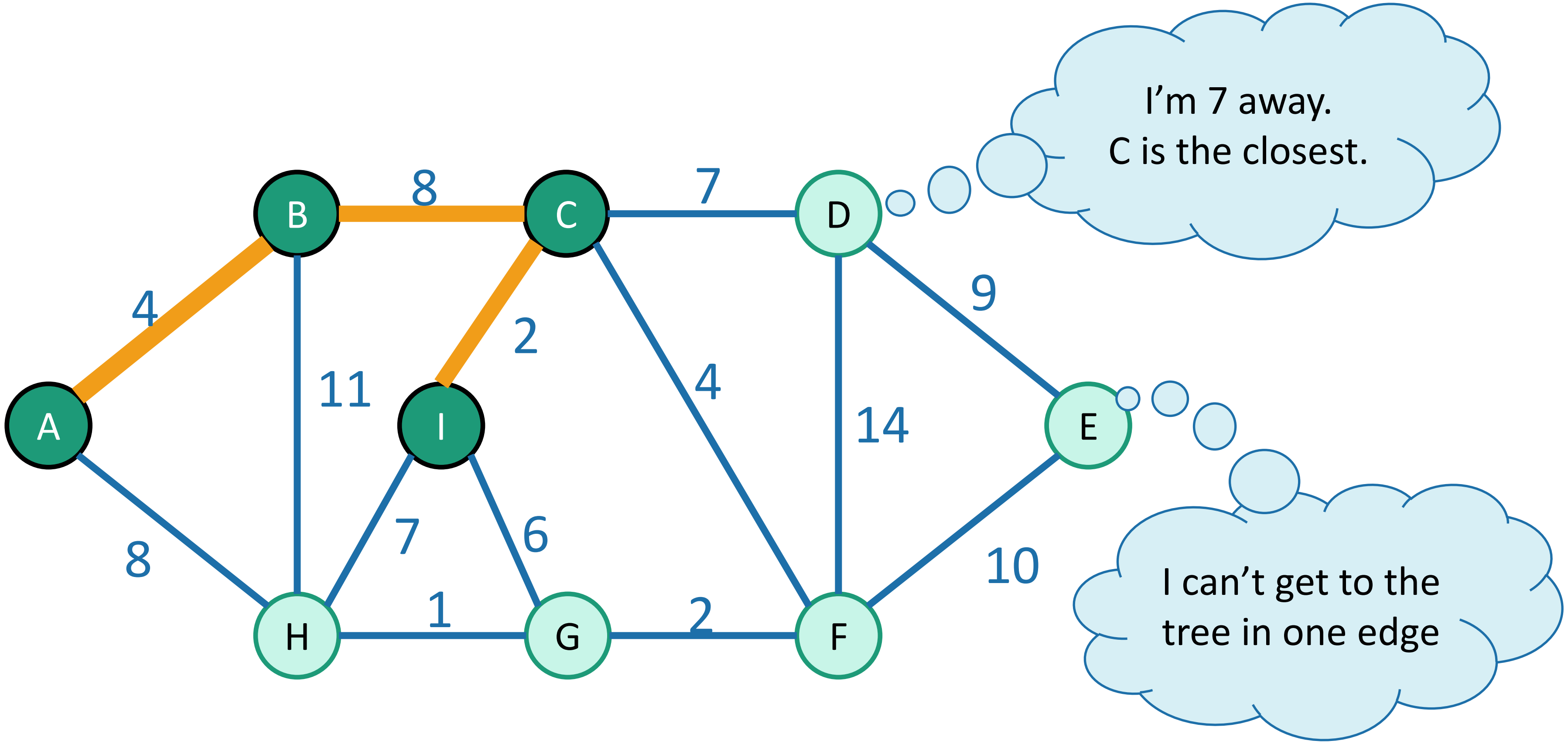
1. Does it work?
   - That is, does it actually return a MST?
     - **Yes!**

2. How do we actually implement this?
   - the pseudocode above says “slowPrim”…

How do we actually implement this?

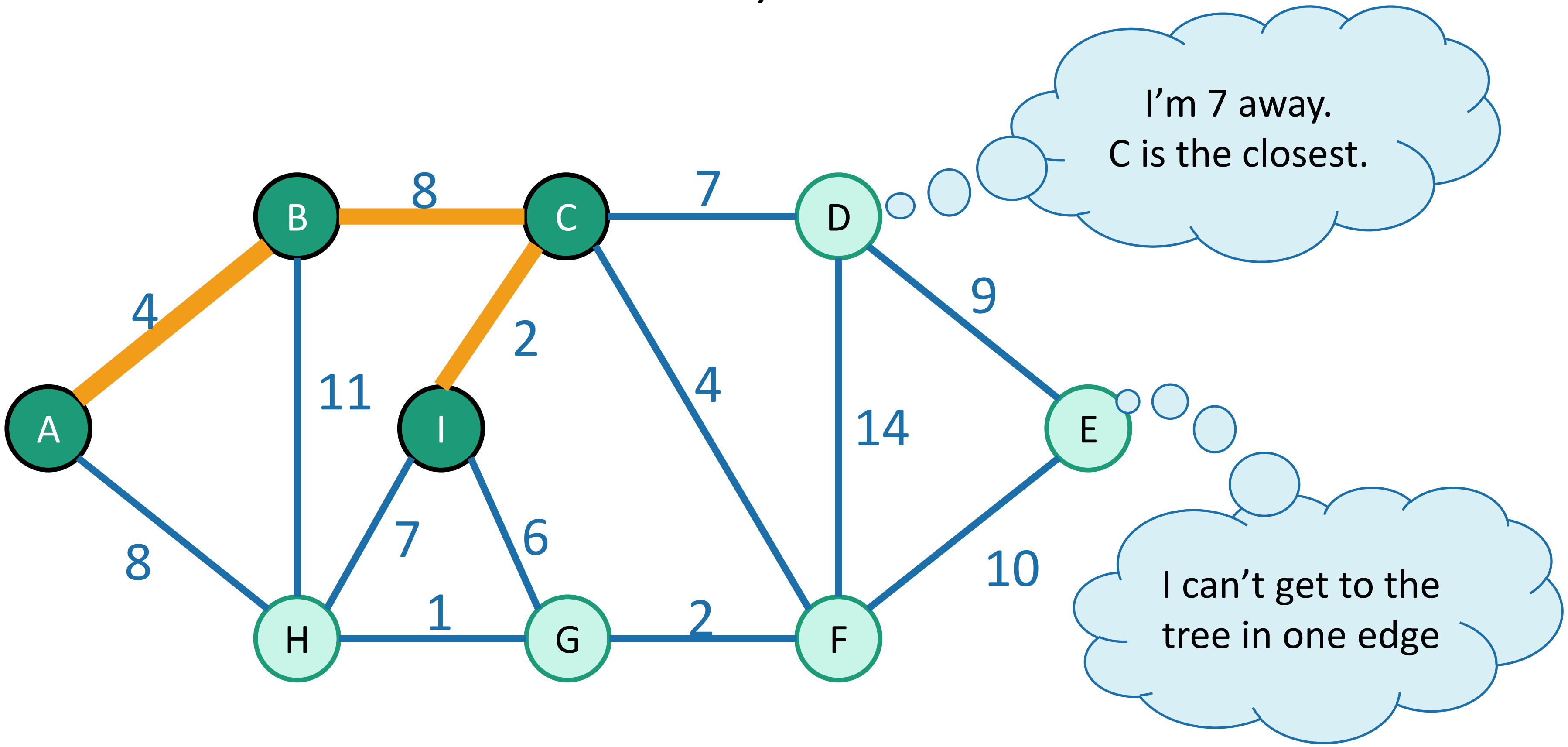
- Each vertex keeps:
  - the **distance** from itself to the **growing spanning tree**
  - **how to get there**.

if you can get there in one edge.



How do we actually implement this?

- Each vertex keeps:
  - the **distance** from itself to the **growing spanning tree** if you can get there in one edge.
  - **how to get there**.
- Choose the closest vertex, add it.

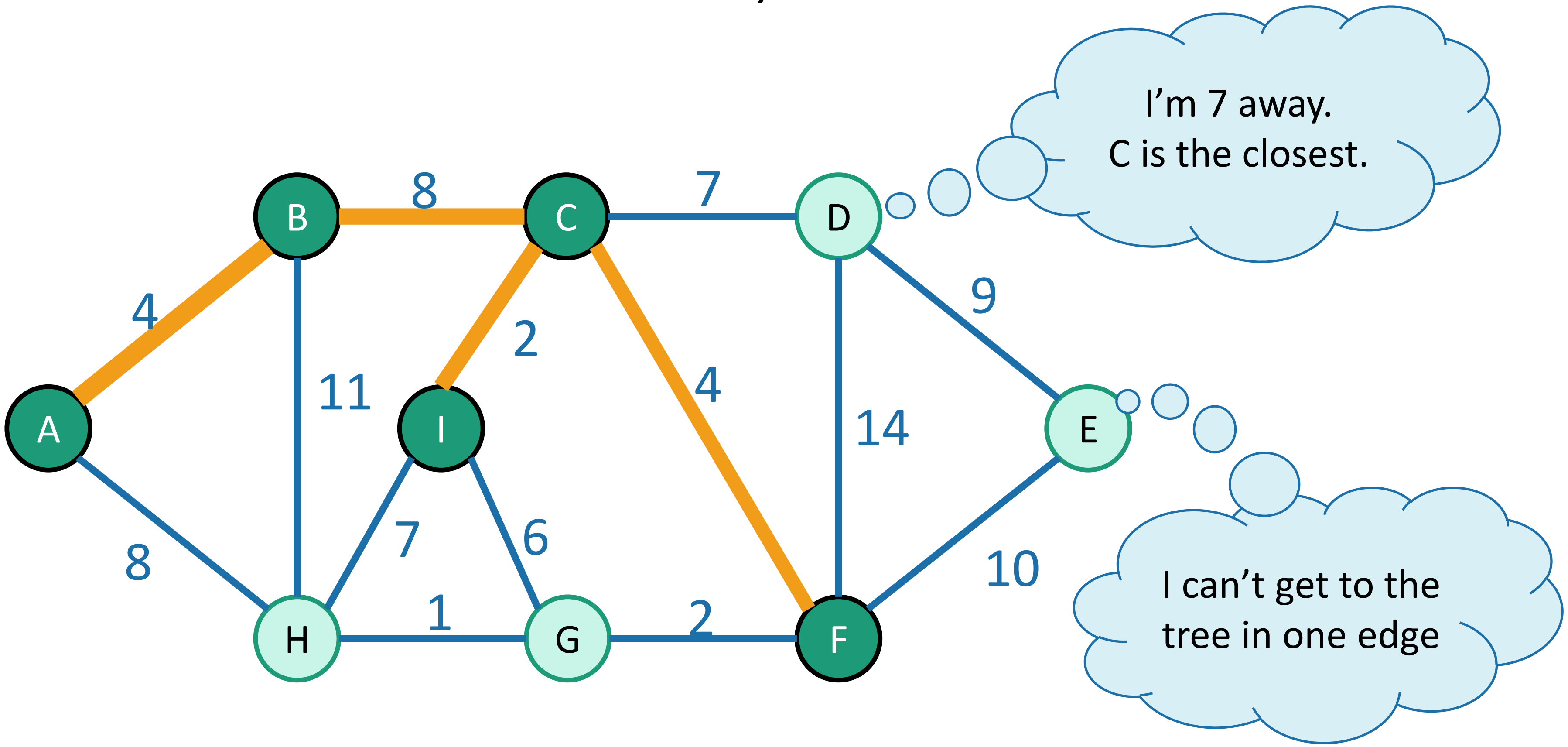


How do we actually implement this?

- Each vertex keeps:
  - the **distance** from itself to the **growing spanning tree**
  - **how to get there**.

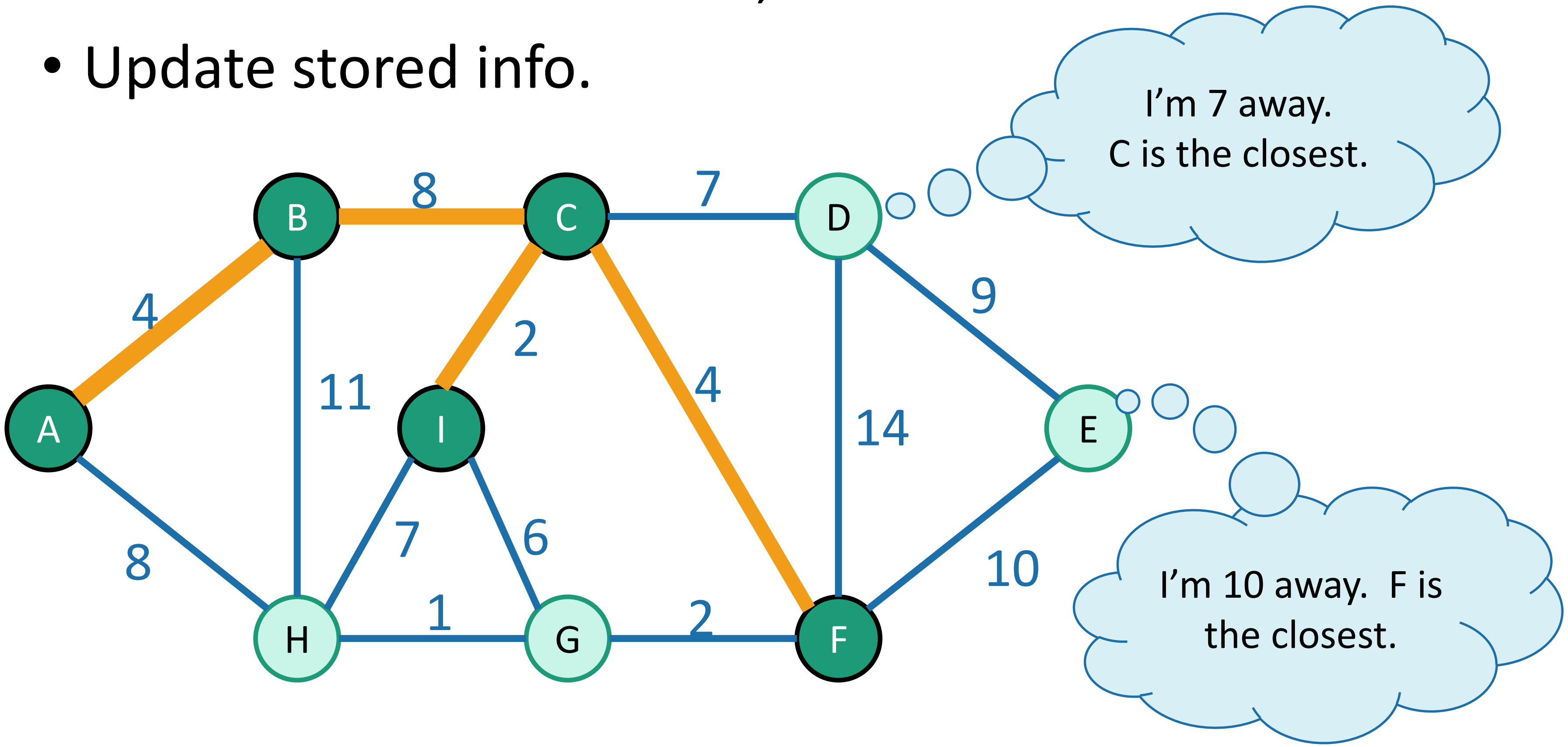
if you can get there in one edge.

- Choose the closest vertex, add it.



How do we actually implement this?

- Each vertex keeps:
  - the **distance** from itself to the **growing spanning tree** if you can get there in one edge.
  - **how to get there**.
- Choose the closest vertex, add it.
- Update stored info.



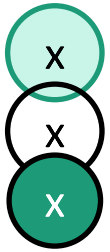
```
MST-PRIM(G, w, r)
for each u ∈ G.V
    u.key = ∞
    u.π = NIL
r.key = 0
Q = G.V
while Q ≠ ∅
    u = EXTRACT-MIN(Q)
    for each v ∈ G.Adj[u]
        if v ∈ Q and w(u, v) < v.key
            v.π = u
            v.key = w(u, v)
```

Pseudocode

Efficient implementation

Every vertex has a key and a parent

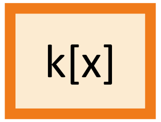
Until all the vertices are **reached:**



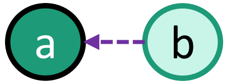
Can't reach x yet

x is "active"

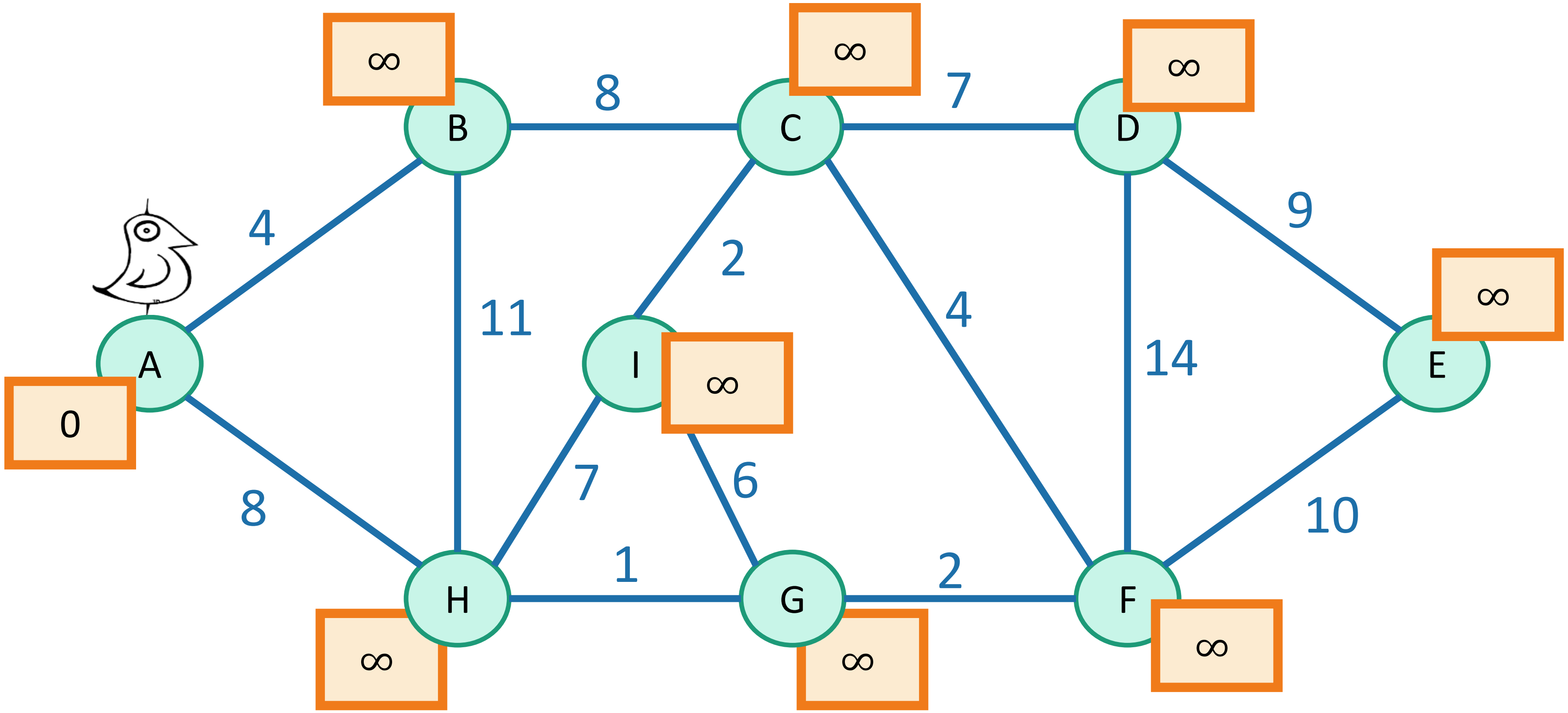
Can reach x



k[x] is the distance of x from the growing tree



p[b] = a, meaning that a was the vertex that k[b] comes from.

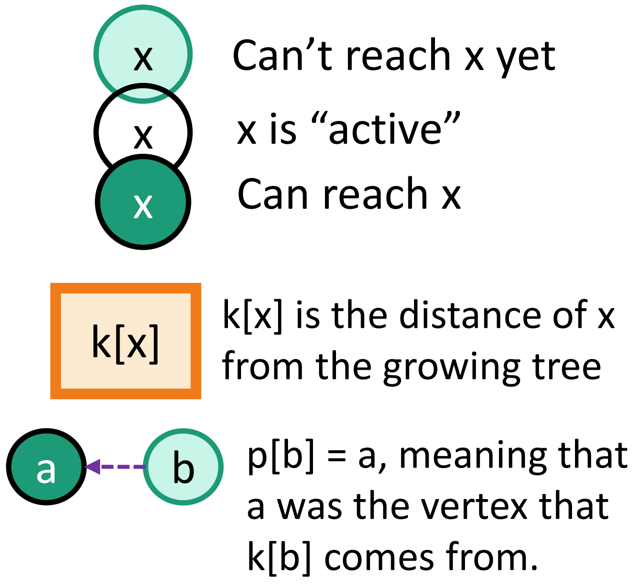


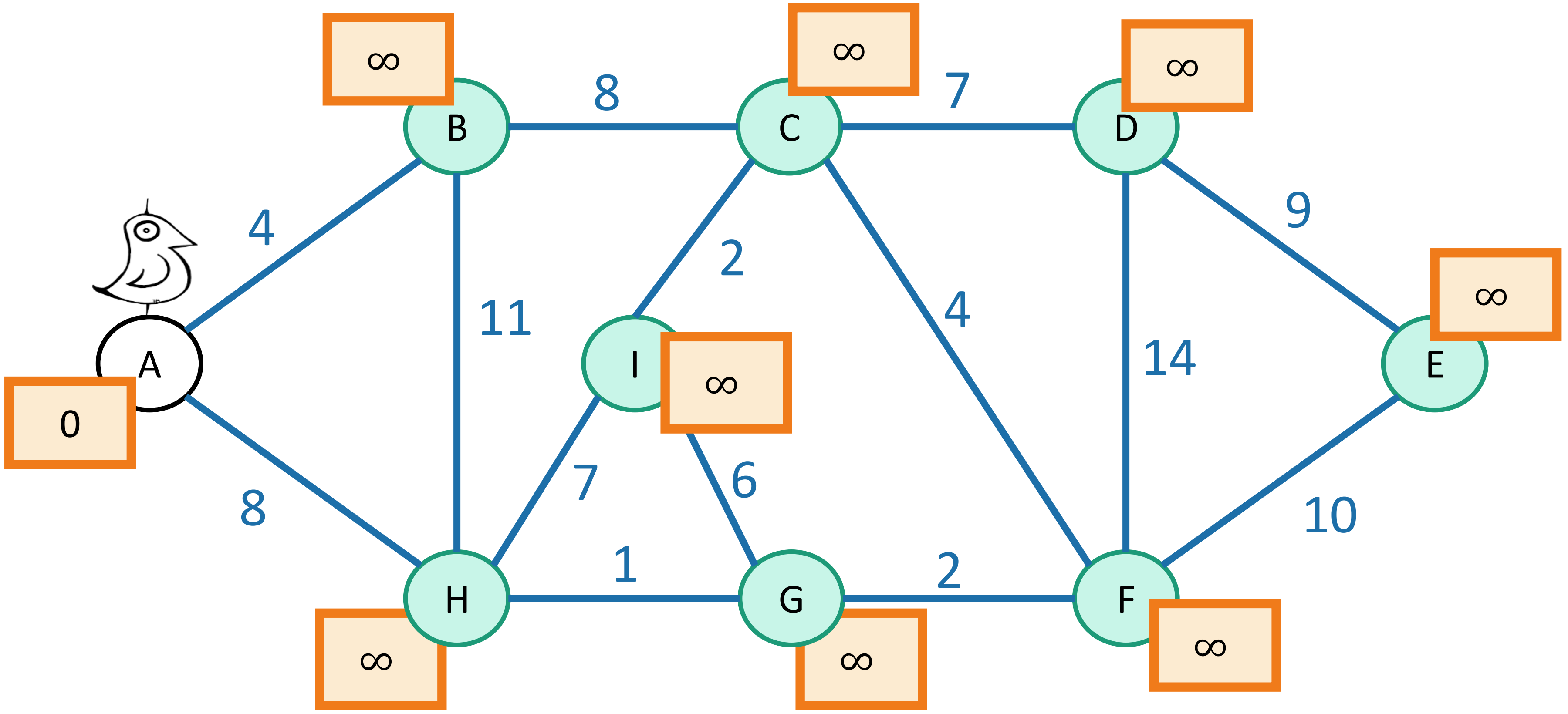
Efficient implementation

Every vertex has a key and a parent

Until all the vertices are **reached:**

- Activate the **unreached** vertex u with the smallest key.



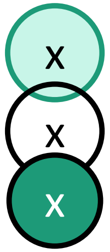


Efficient implementation

Every vertex has a key and a parent

Until all the vertices are **reached:**

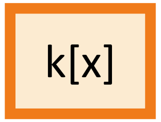
- Activate the **unreached** vertex u with the smallest key.
- **for each** of u's neighbors v:
  - k[v] = min(k[v], weight(u,v))
  - if k[v] updated, p[v] = u



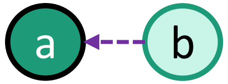
Can't reach x yet

x is "active"

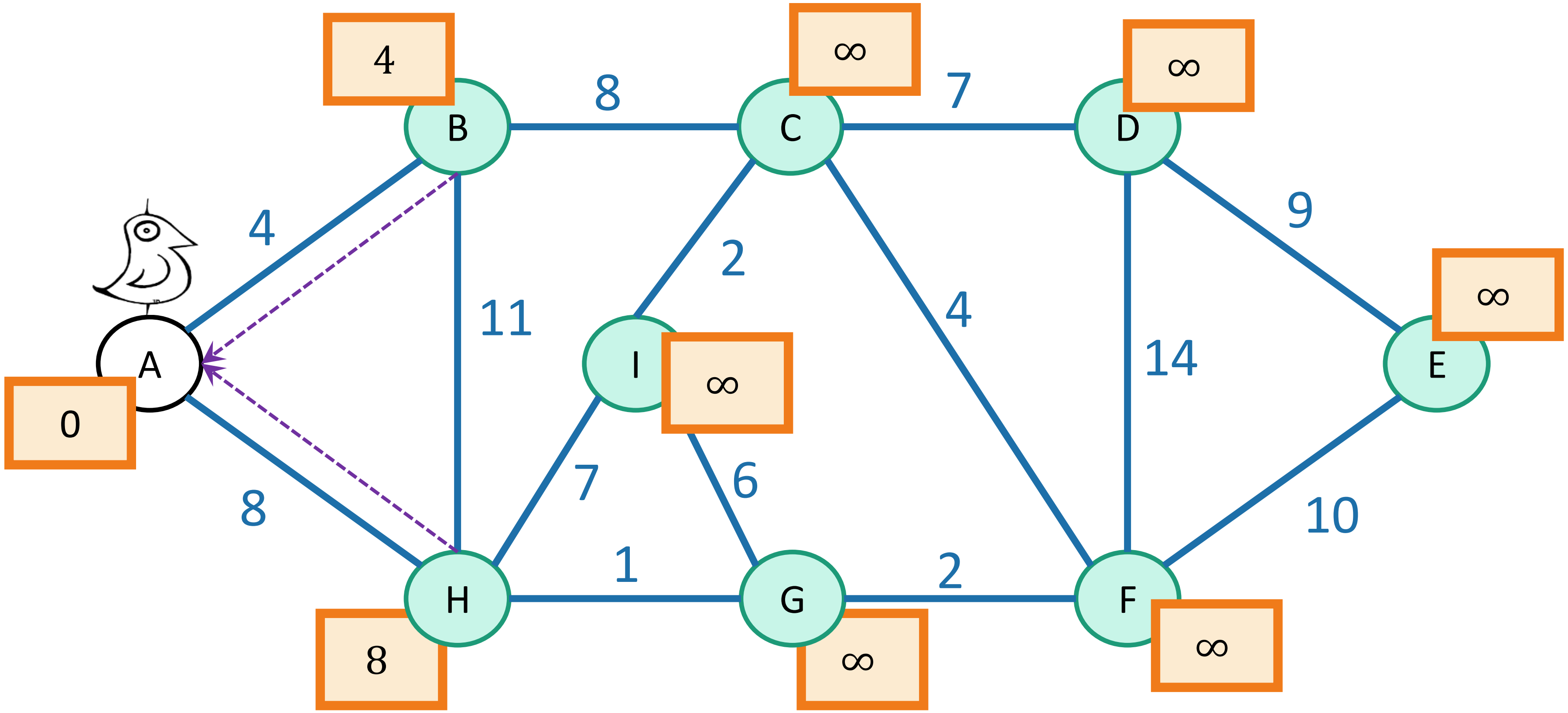
Can reach x



k[x] is the distance of x from the growing tree



p[b] = a, meaning that a was the vertex that k[b] comes from.

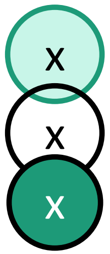


Efficient implementation

Every vertex has a key and a parent

Until all the vertices are **reached:**

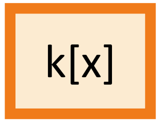
- Activate the **unreached** vertex u with the smallest key.
- **for each** of u's neighbors v:
  - k[v] = min(k[v], weight(u,v))
  - if k[v] updated, p[v] = u
- Mark u as **reached,** and **add (p[u],u) to MST.**



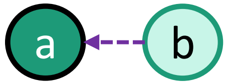
Can't reach x yet

x is "active"

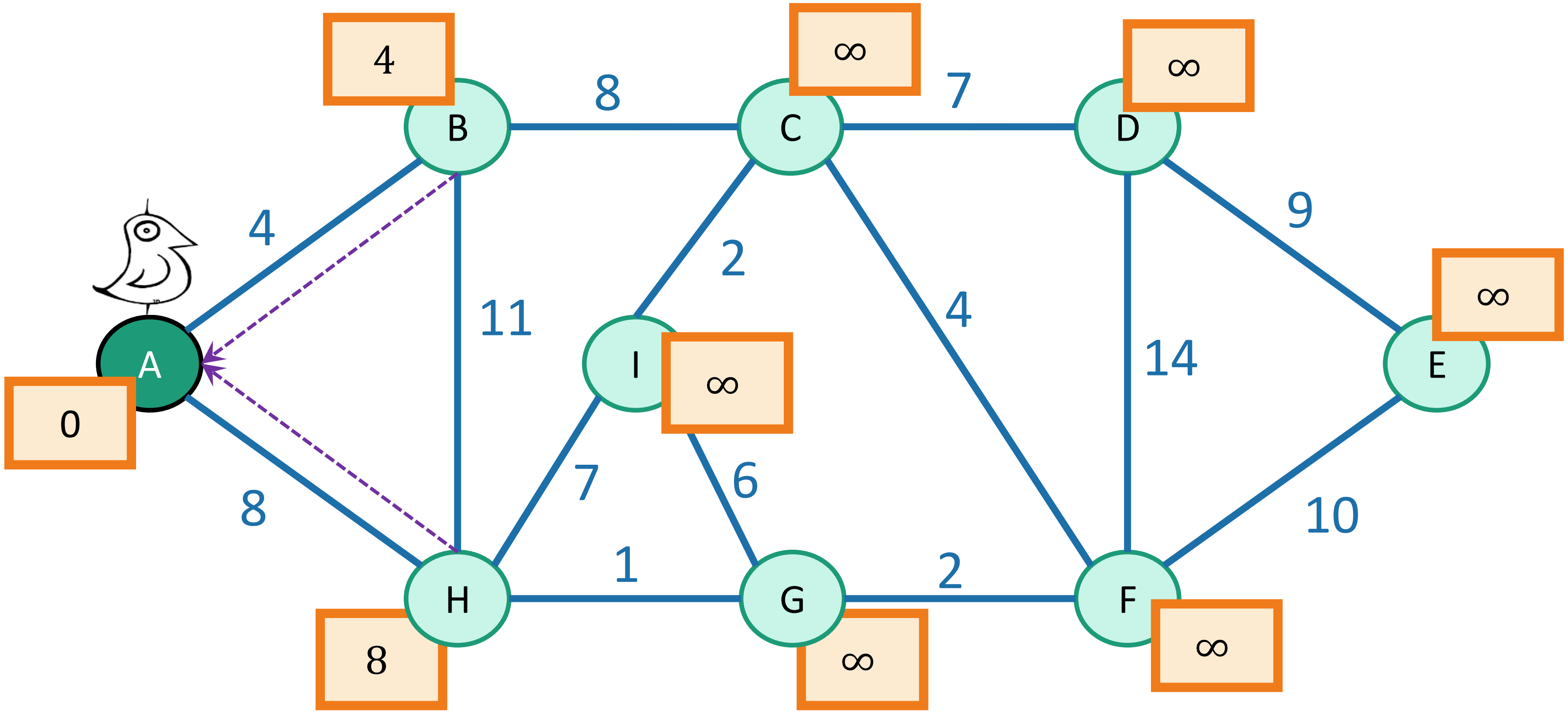
Can reach x



k[x] is the distance of x from the growing tree



p[b] = a, meaning that a was the vertex that k[b] comes from.

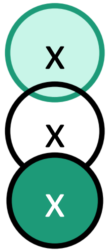


Efficient implementation

Every vertex has a key and a parent

Until all the vertices are **reached:**

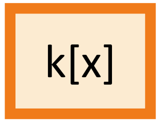
- Activate the **unreached** vertex u with the smallest key.
- **for each** of u's neighbors v:
  - k[v] = min(k[v], weight(u,v))
  - if k[v] updated, p[v] = u
- Mark u as **reached,** and **add (p[u],u) to MST.**



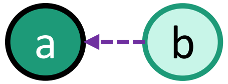
Can't reach x yet

x is "active"

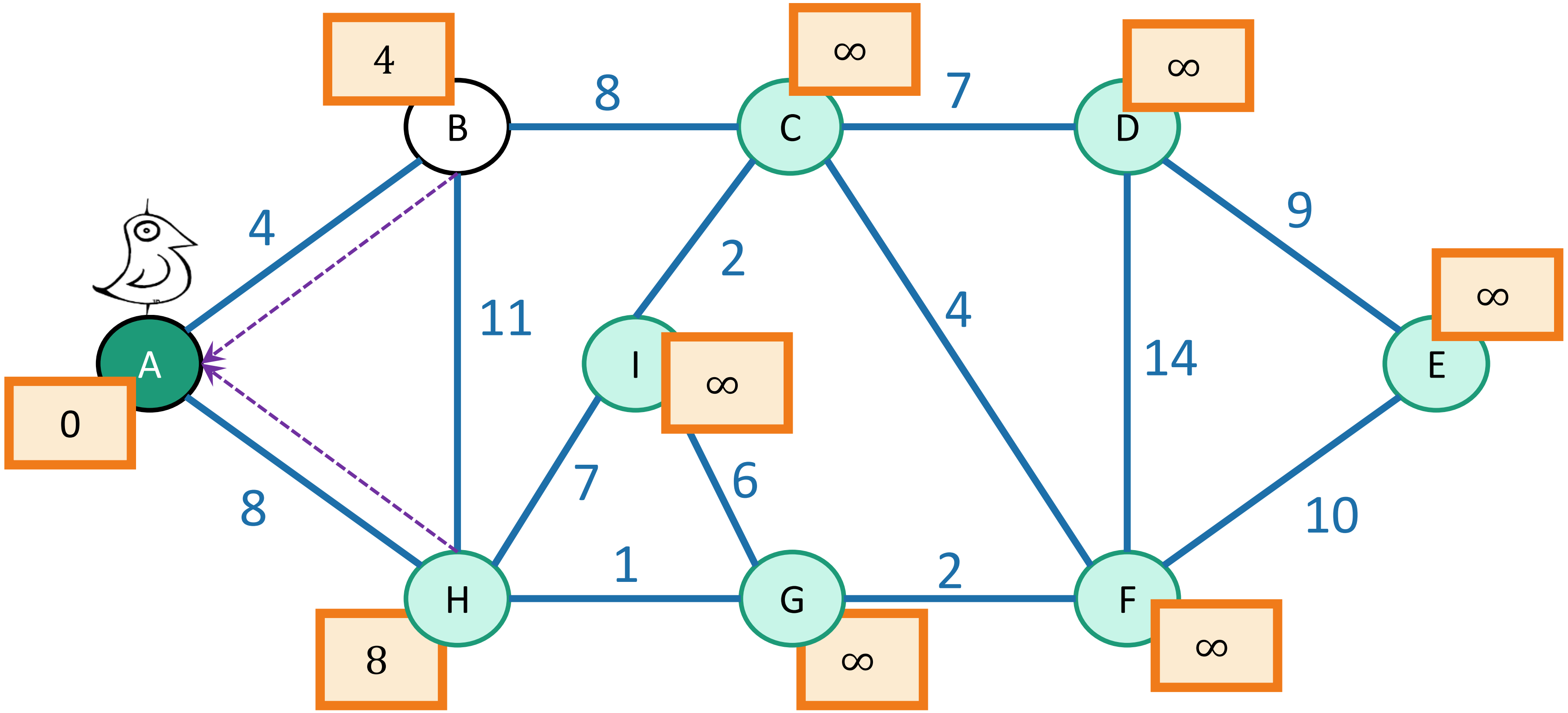
Can reach x



k[x] is the distance of x from the growing tree



p[b] = a, meaning that a was the vertex that k[b] comes from.

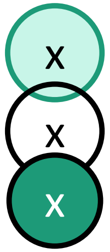


Efficient implementation

Every vertex has a key and a parent

Until all the vertices are **reached:**

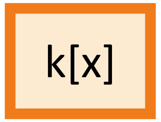
- Activate the **unreached** vertex u with the smallest key.
- **for each** of u's neighbors v:
  - k[v] = min(k[v], weight(u,v))
  - if k[v] updated, p[v] = u
- Mark u as **reached,** and **add (p[u],u) to MST.**



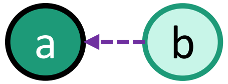
Can't reach x yet

x is "active"

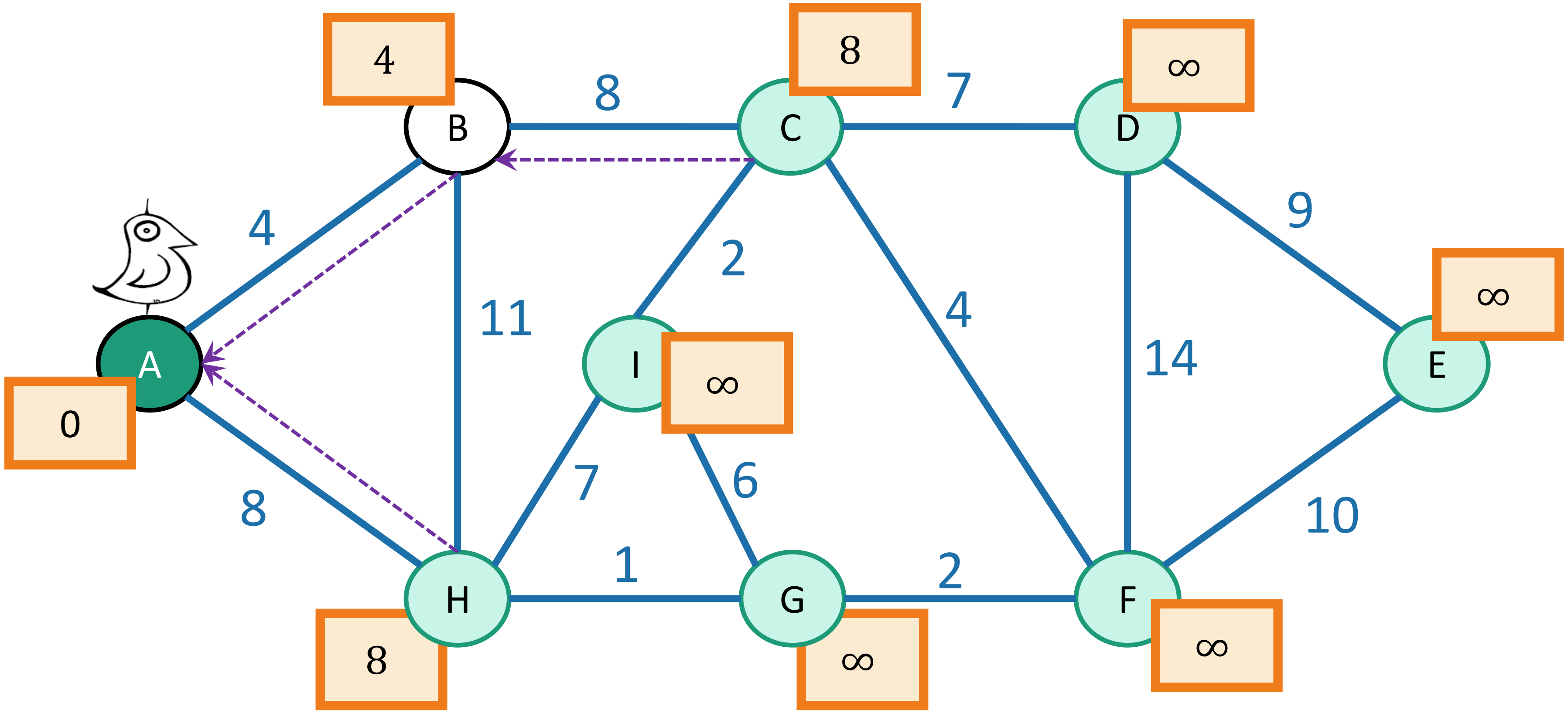
Can reach x



k[x] is the distance of x from the growing tree



p[b] = a, meaning that a was the vertex that k[b] comes from.

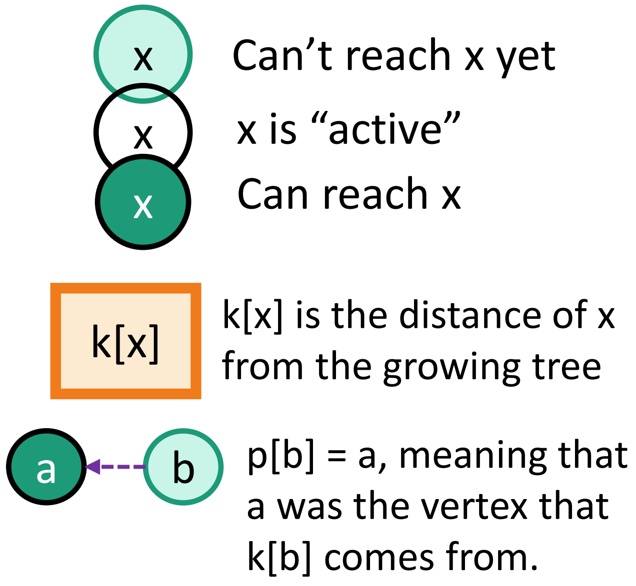


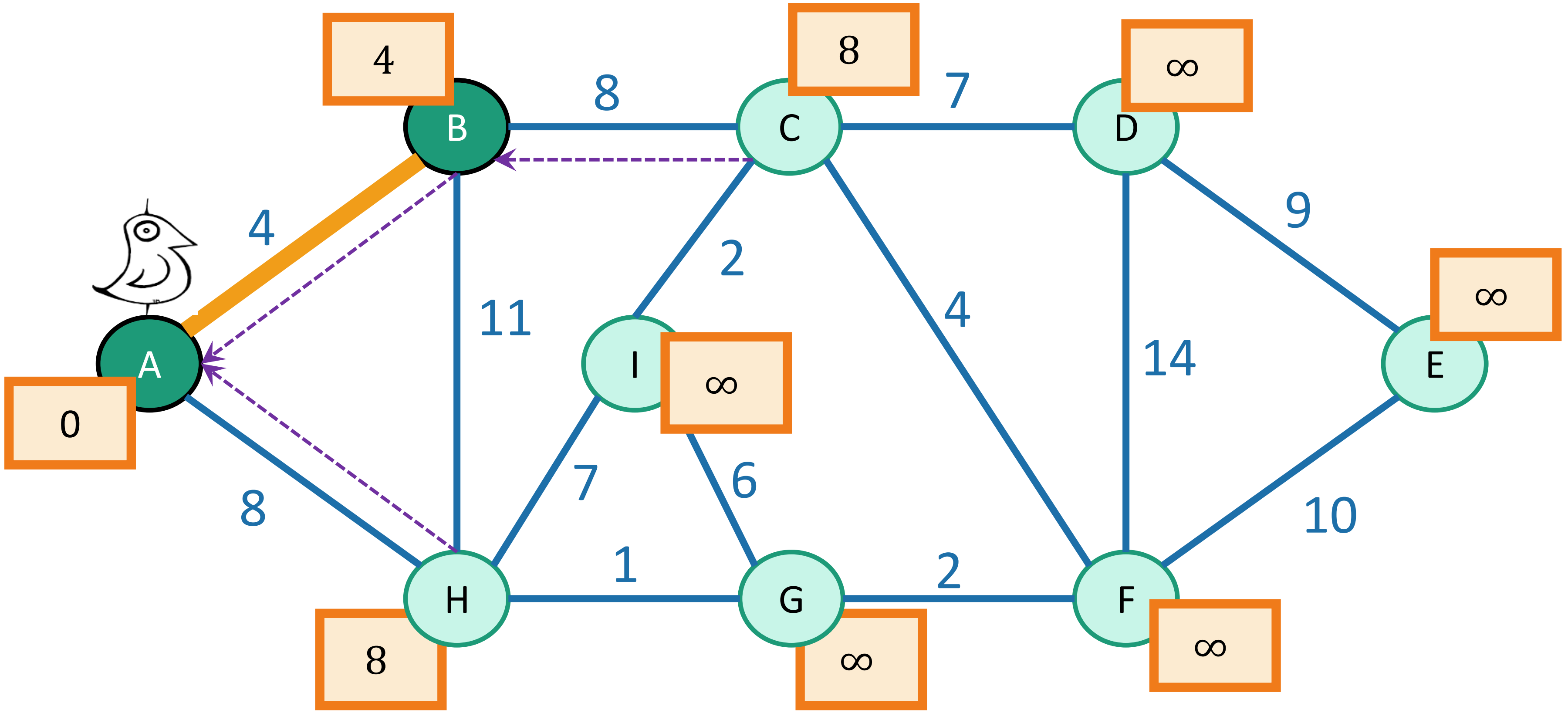
Efficient implementation

Every vertex has a key and a parent

Until all the vertices are **reached:**

- Activate the **unreached** vertex u with the smallest key.
- **for each** of u's neighbors v:
  - k[v] = min(k[v], weight(u,v))
  - if k[v] updated, p[v] = u
- Mark u as **reached,** and **add (p[u],u) to MST.**



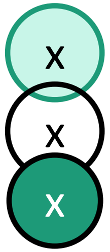


Efficient implementation

Every vertex has a key and a parent

Until all the vertices are **reached:**

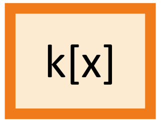
- Activate the **unreached** vertex u with the smallest key.
- **for each** of u's neighbors v:
  - k[v] = min(k[v], weight(u,v))
  - if k[v] updated, p[v] = u
- Mark u as **reached,** and **add (p[u],u) to MST.**



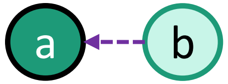
Can't reach x yet

x is "active"

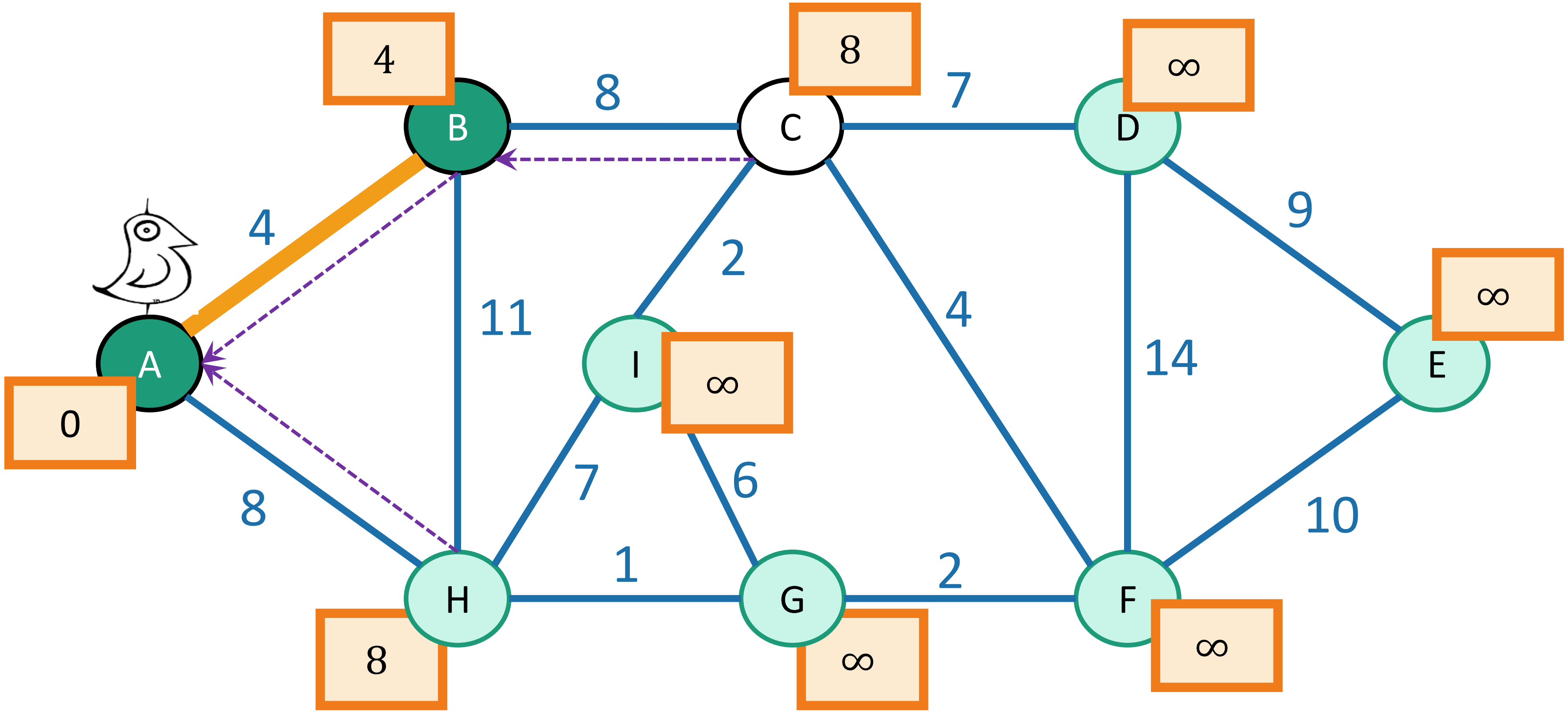
Can reach x



k[x] is the distance of x from the growing tree



p[b] = a, meaning that a was the vertex that k[b] comes from.

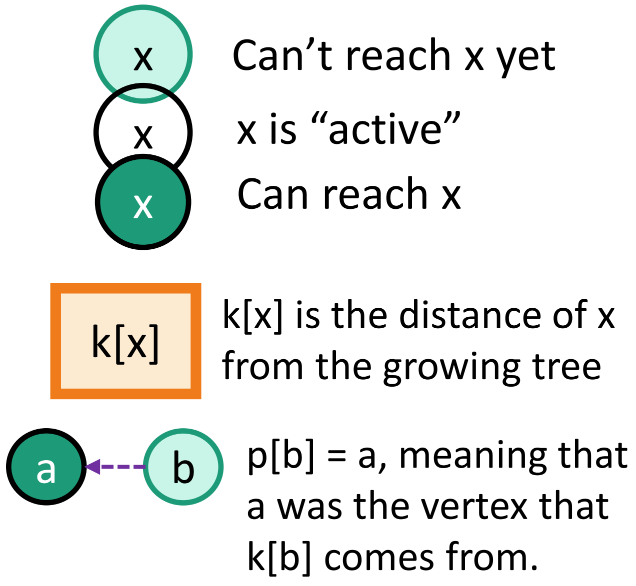


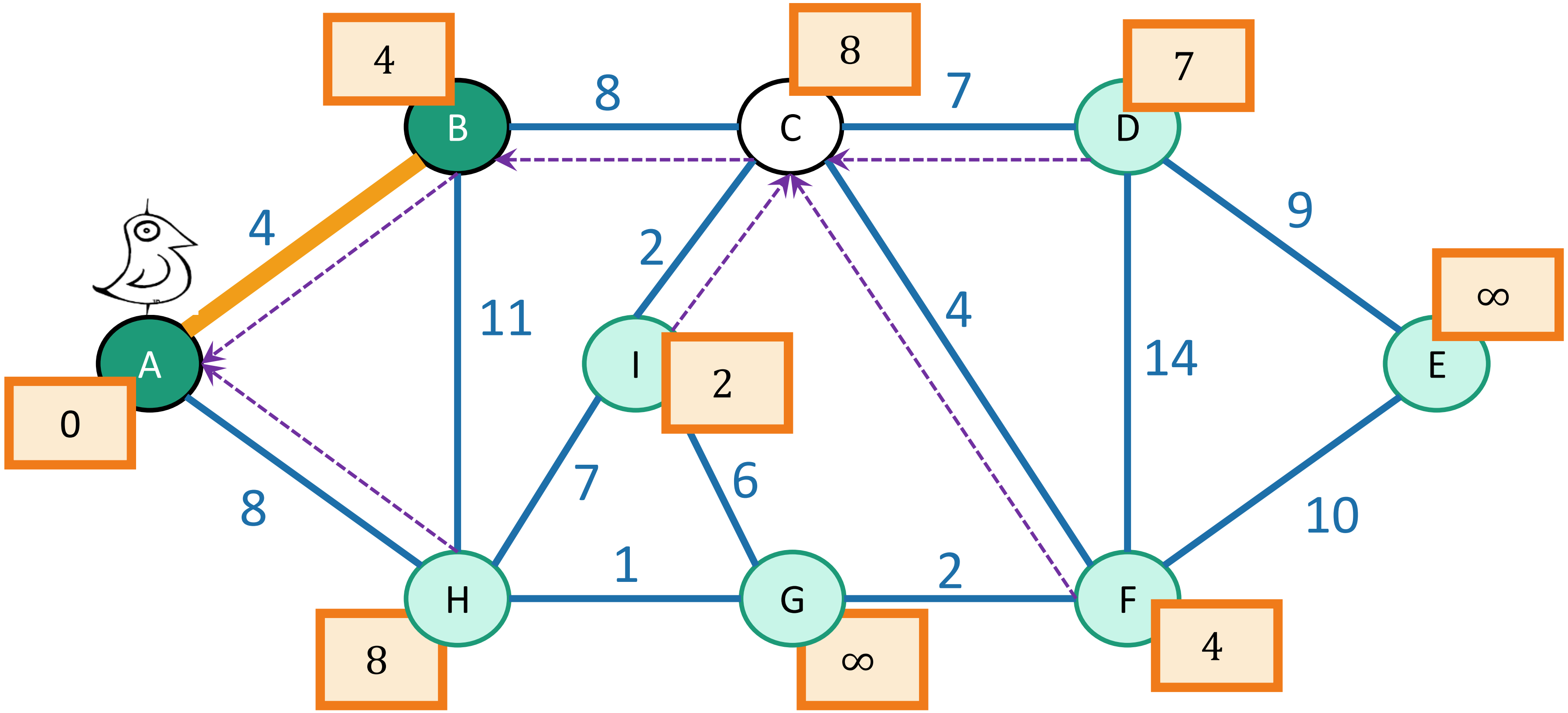
Efficient implementation

Every vertex has a key and a parent

Until all the vertices are **reached:**

- Activate the **unreached** vertex u with the smallest key.
- **for each** of u's neighbors v:
  - k[v] = min(k[v], weight(u,v))
  - if k[v] updated, p[v] = u
- Mark u as **reached,** and **add (p[u],u) to MST.**



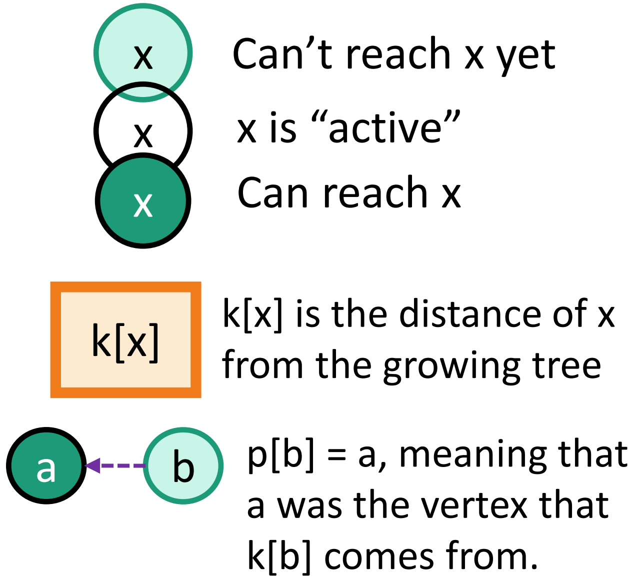


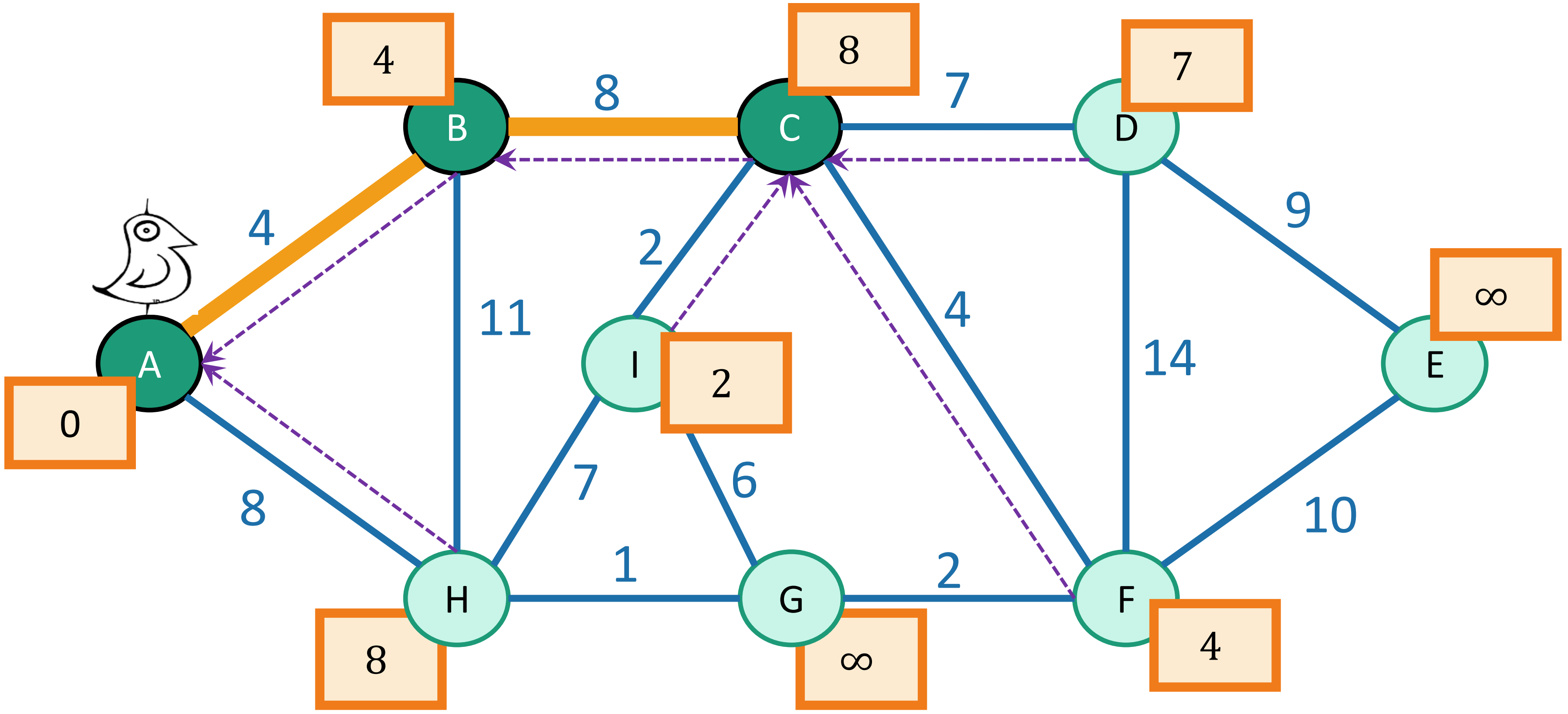
Efficient implementation

Every vertex has a key and a parent

Until all the vertices are **reached:**

- Activate the **unreached** vertex u with the smallest key.
- **for each** of u's neighbors v:
  - k[v] = min(k[v], weight(u,v))
  - if k[v] updated, p[v] = u
- Mark u as **reached,** and **add (p[u],u) to MST.**



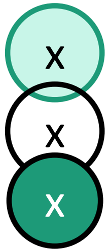


Efficient implementation

Every vertex has a key and a parent

Until all the vertices are **reached:**

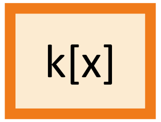
- Activate the **unreached** vertex u with the smallest key.
- **for each** of u's neighbors v:
  - k[v] = min(k[v], weight(u,v))
  - if k[v] updated, p[v] = u
- Mark u as **reached,** and **add (p[u],u) to MST.**



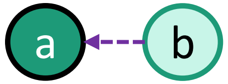
Can't reach x yet

x is "active"

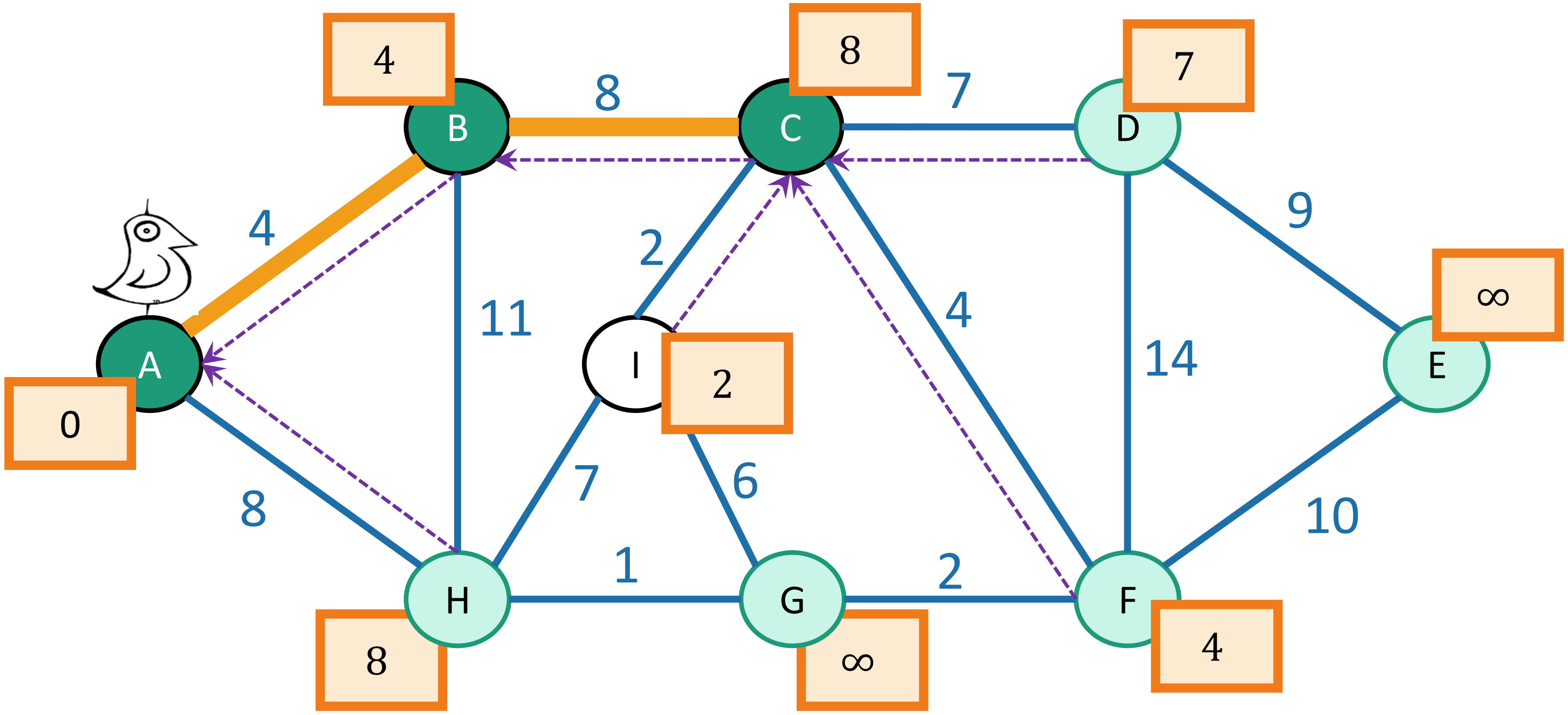
Can reach x



k[x] is the distance of x from the growing tree



p[b] = a, meaning that a was the vertex that k[b] comes from.

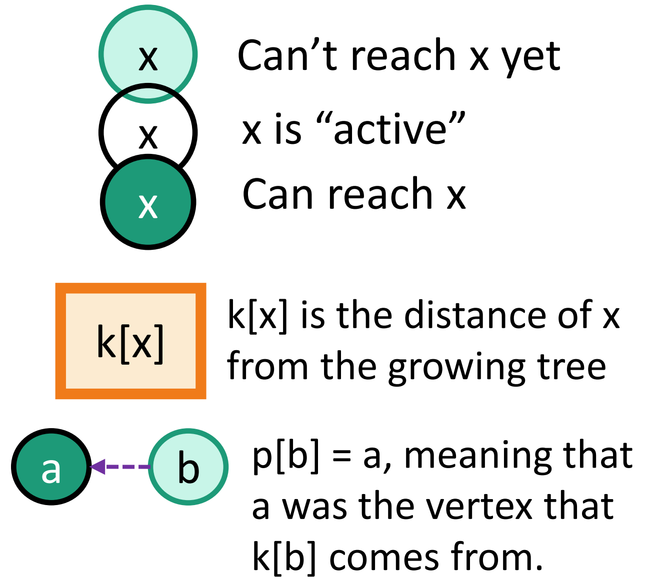


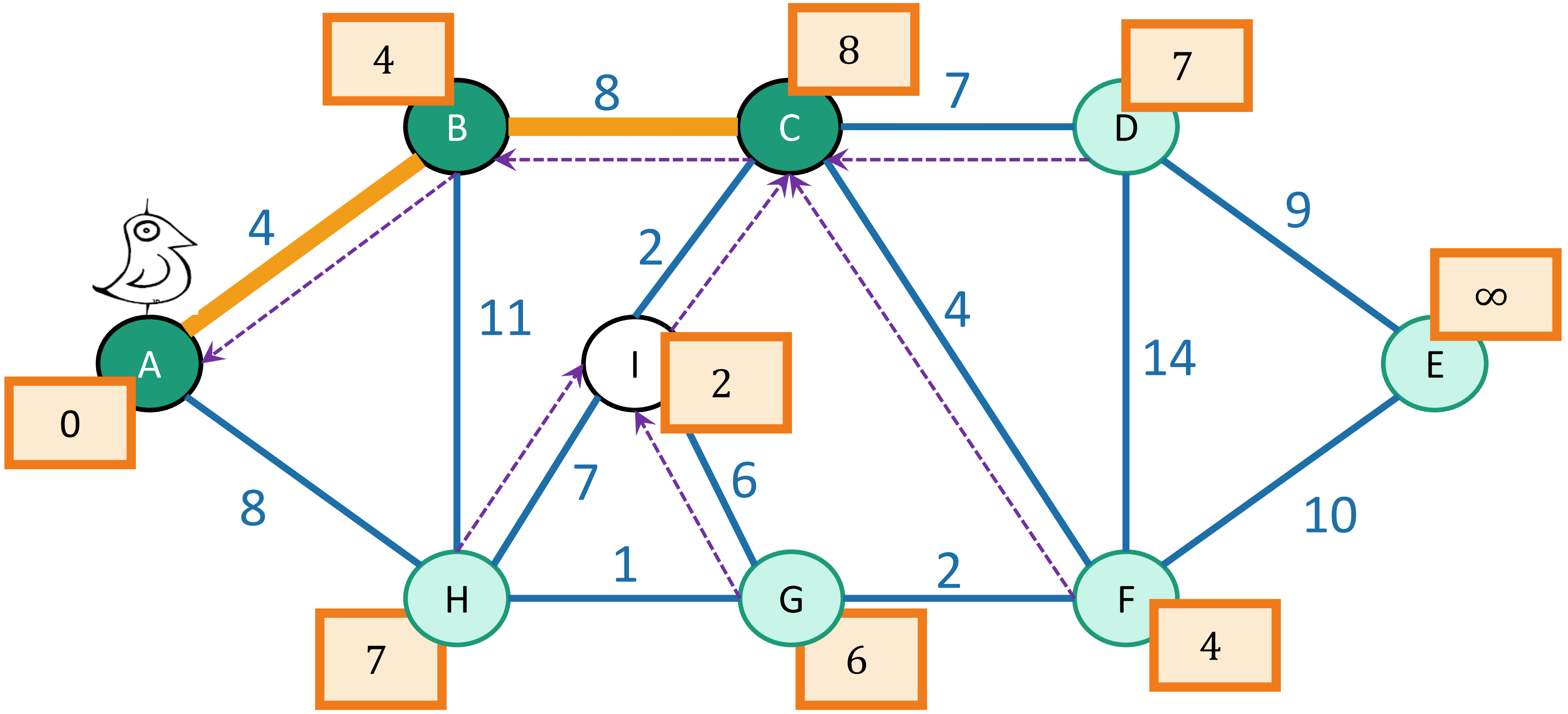
Efficient implementation

Every vertex has a key and a parent

Until all the vertices are **reached:**

- Activate the **unreached** vertex u with the smallest key.
- **for each** of u's neighbors v:
    - k[v] = min(k[v], weight(u,v))
    - if k[v] updated, p[v] = u
- Mark u as **reached,** and **add (p[u],u) to MST.**



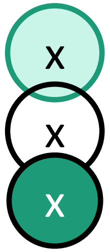


Efficient implementation

Every vertex has a key and a parent

Until all the vertices are **reached:**

- Activate the **unreached** vertex u with the smallest key.
- **for each** of u's neighbors v:
  - k[v] = min(k[v], weight(u,v))
  - if k[v] updated, p[v] = u
- Mark u as **reached,** and **add (p[u],u) to MST.**

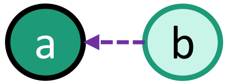


Can't reach x yet

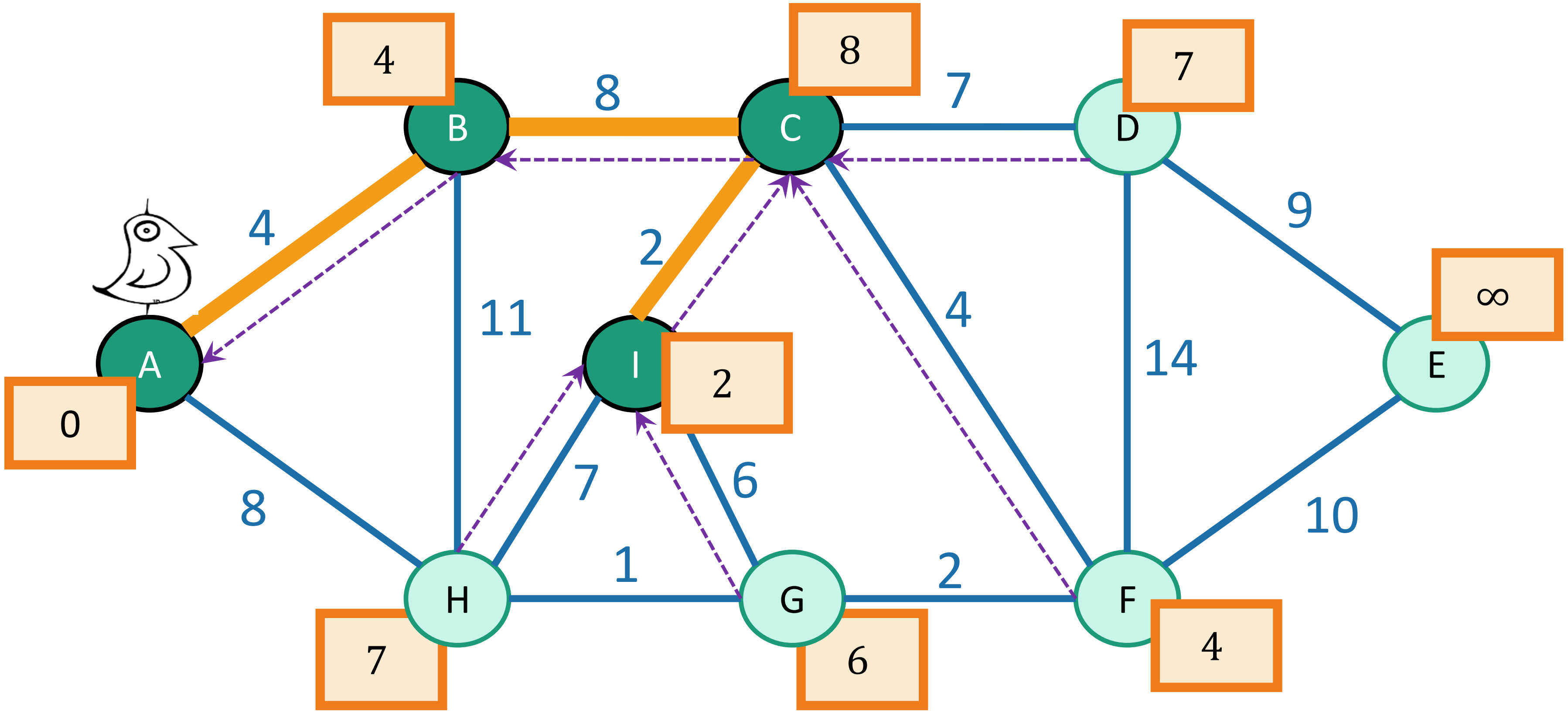
x is "active"

Can reach x

k[x] — k[x] is the distance of x from the growing tree



p[b] = a, meaning that a was the vertex that k[b] comes from.

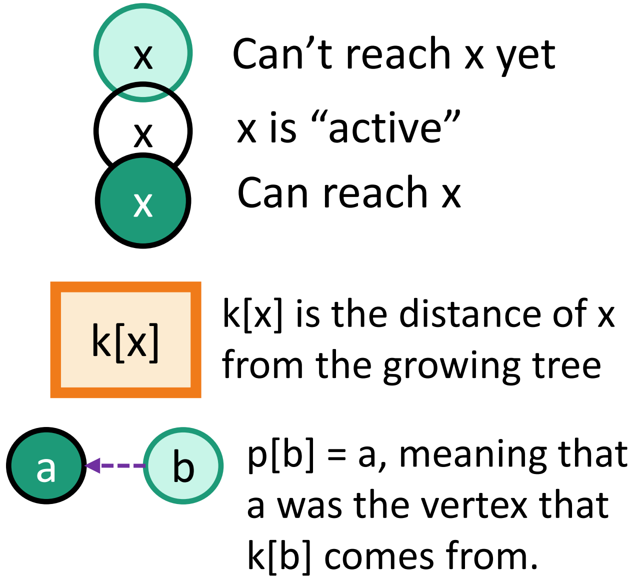


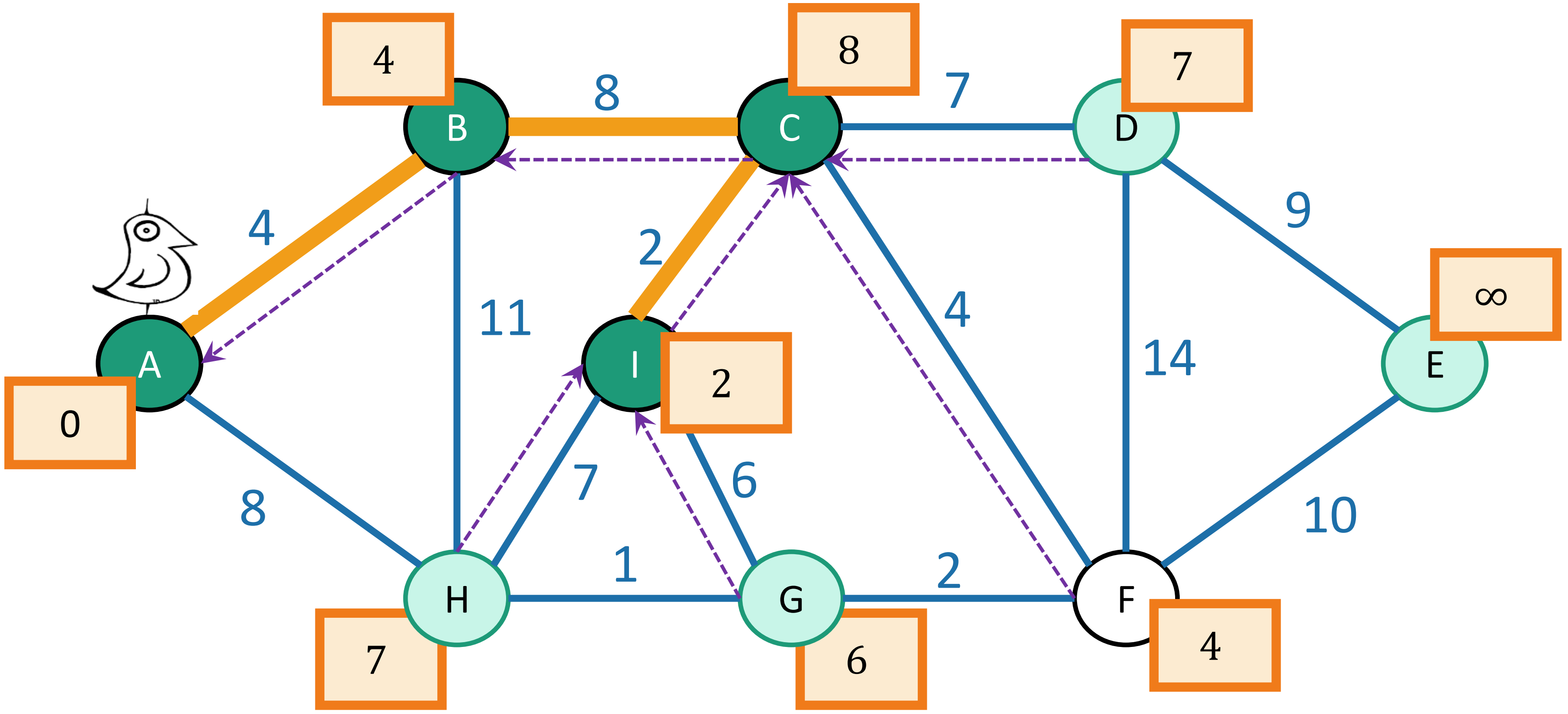
Efficient implementation

Every vertex has a key and a parent

Until all the vertices are **reached:**

- Activate the **unreached** vertex u with the smallest key.
- **for each** of u's neighbors v:
  - k[v] = min(k[v], weight(u,v))
  - if k[v] updated, p[v] = u
- Mark u as **reached,** and **add (p[u],u) to MST.**



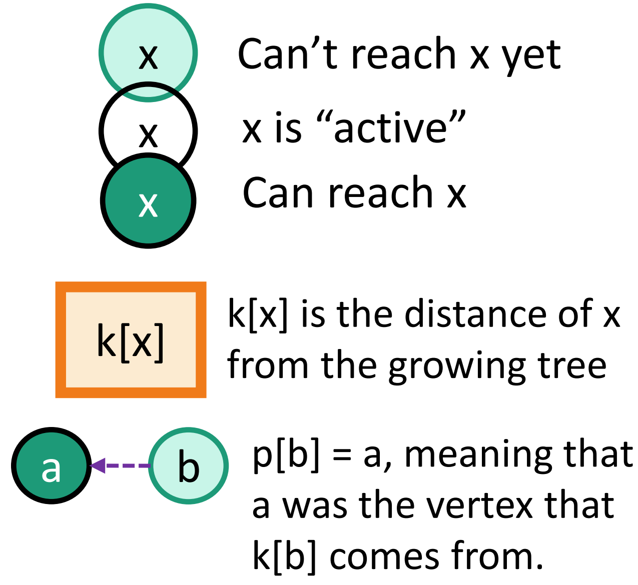


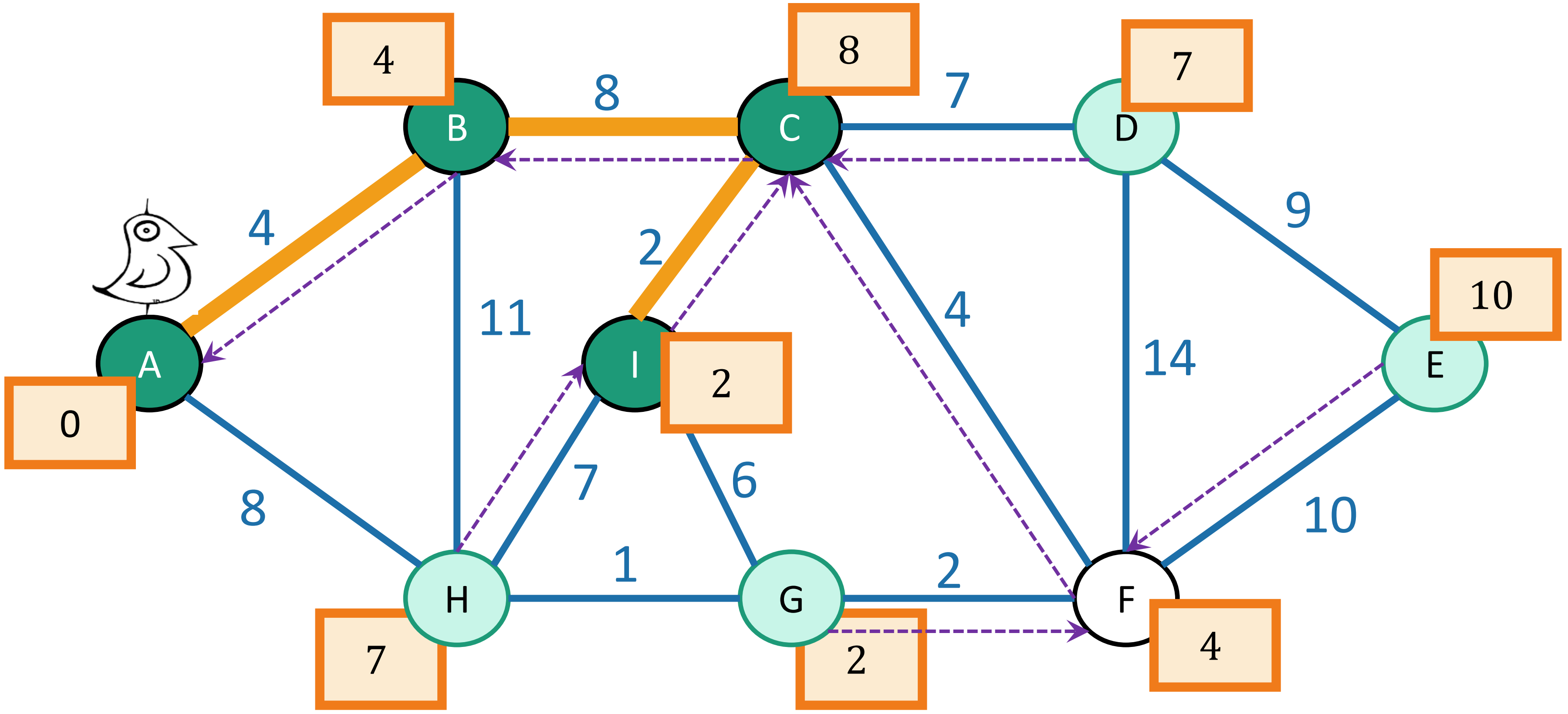
Efficient implementation

Every vertex has a key and a parent

Until all the vertices are **reached:**

- Activate the **unreached** vertex u with the smallest key.
- **for each** of u's neighbors v:
  - k[v] = min(k[v], weight(u,v))
  - if k[v] updated, p[v] = u
- Mark u as **reached,** and **add (p[u],u) to MST.**





Efficient implementation

Every vertex has a key and a parent

Until all the vertices are **reached:**

- Activate the **unreached** vertex u with the smallest key.
- **for each** of u's neighbors v:
  - k[v] = min(k[v], weight(u,v))
  - if k[v] updated, p[v] = u
- Mark u as **reached,** and **add (p[u],u) to MST.**

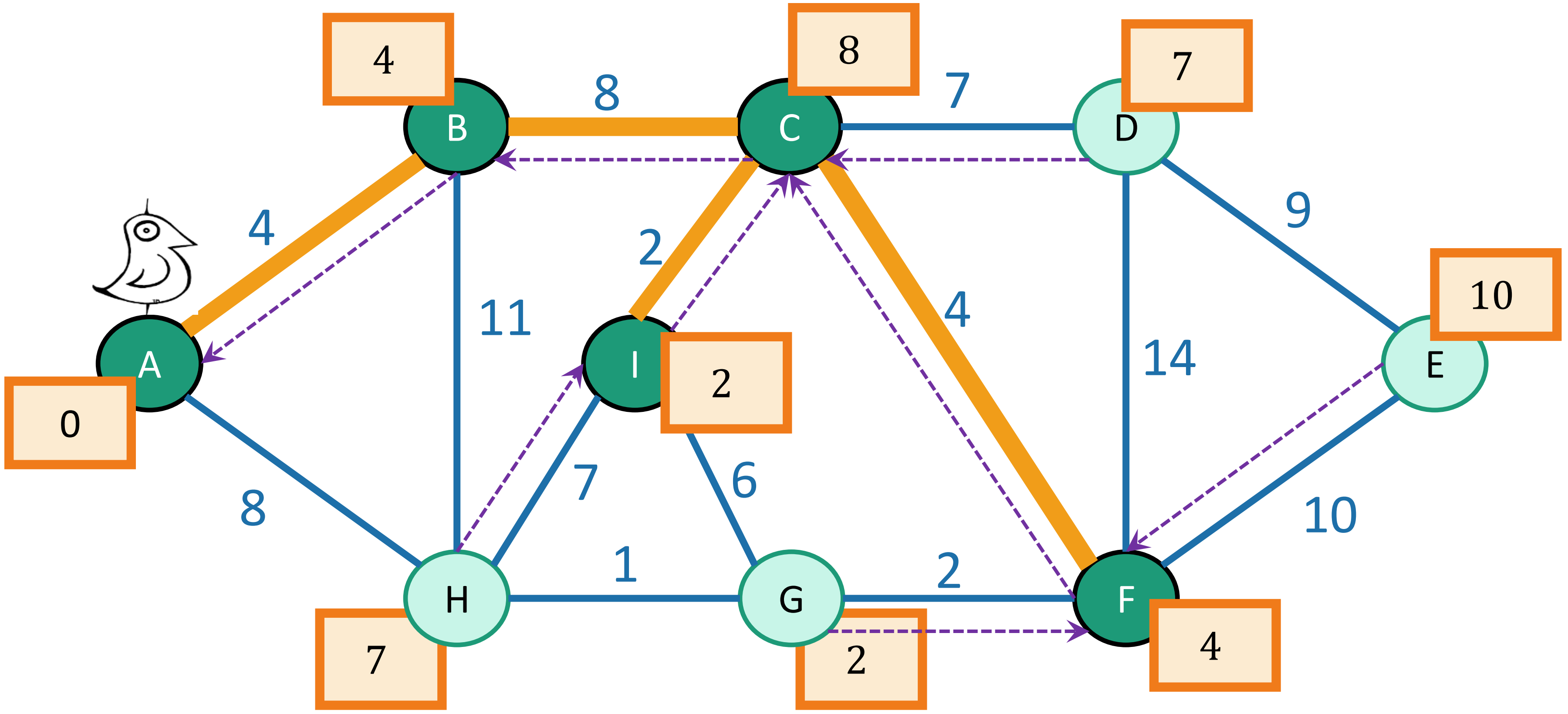
x Can't reach x yet

x x is "active"

x Can reach x

k[x] k[x] is the distance of x from the growing tree

a b p[b] = a, meaning that a was the vertex that k[b] comes from.



Efficient implementation

Every vertex has a key and a parent

x Can't reach x yet

x x is "active"

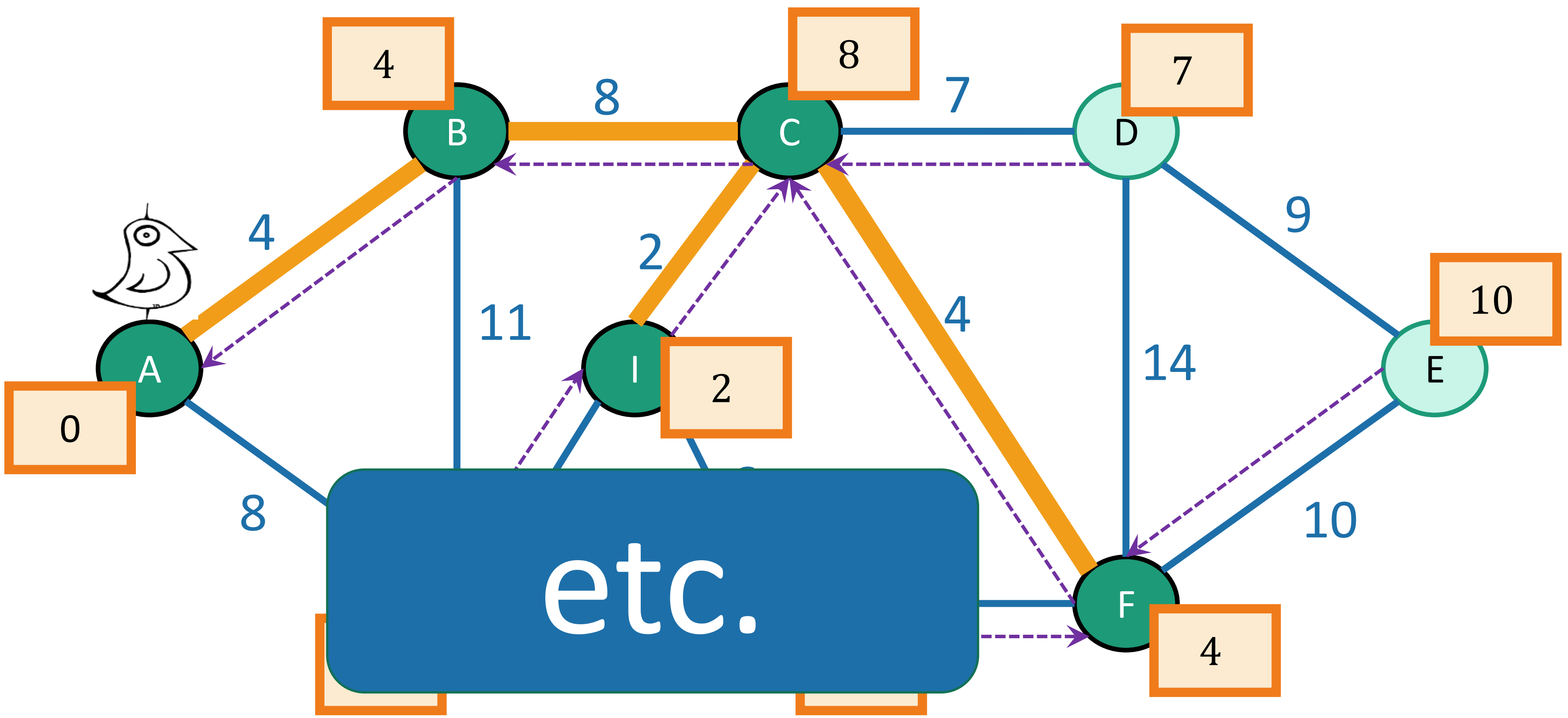
x Can reach x

k[x] k[x] is the distance of x from the growing tree

a b p[b] = a, meaning that a was the vertex that k[b] comes from.

Until all the vertices are **reached:**

- Activate the **unreached** vertex u with the smallest key.
- **for each** of u's neighbors v:
  - k[v] = min(k[v], weight(u,v))
  - if k[v] updated, p[v] = u
- Mark u as **reached,** and **add (p[u],u) to MST.**



One thing that is similar:
Running time

- O(mlog(n)) using a Red-Black tree as a priority queue.

Two questions

1. Does it work?
   - That is, does it actually return a MST?
     - **Yes!**

2. How do we actually implement this?
   - the pseudocode above says “slowPrim”…

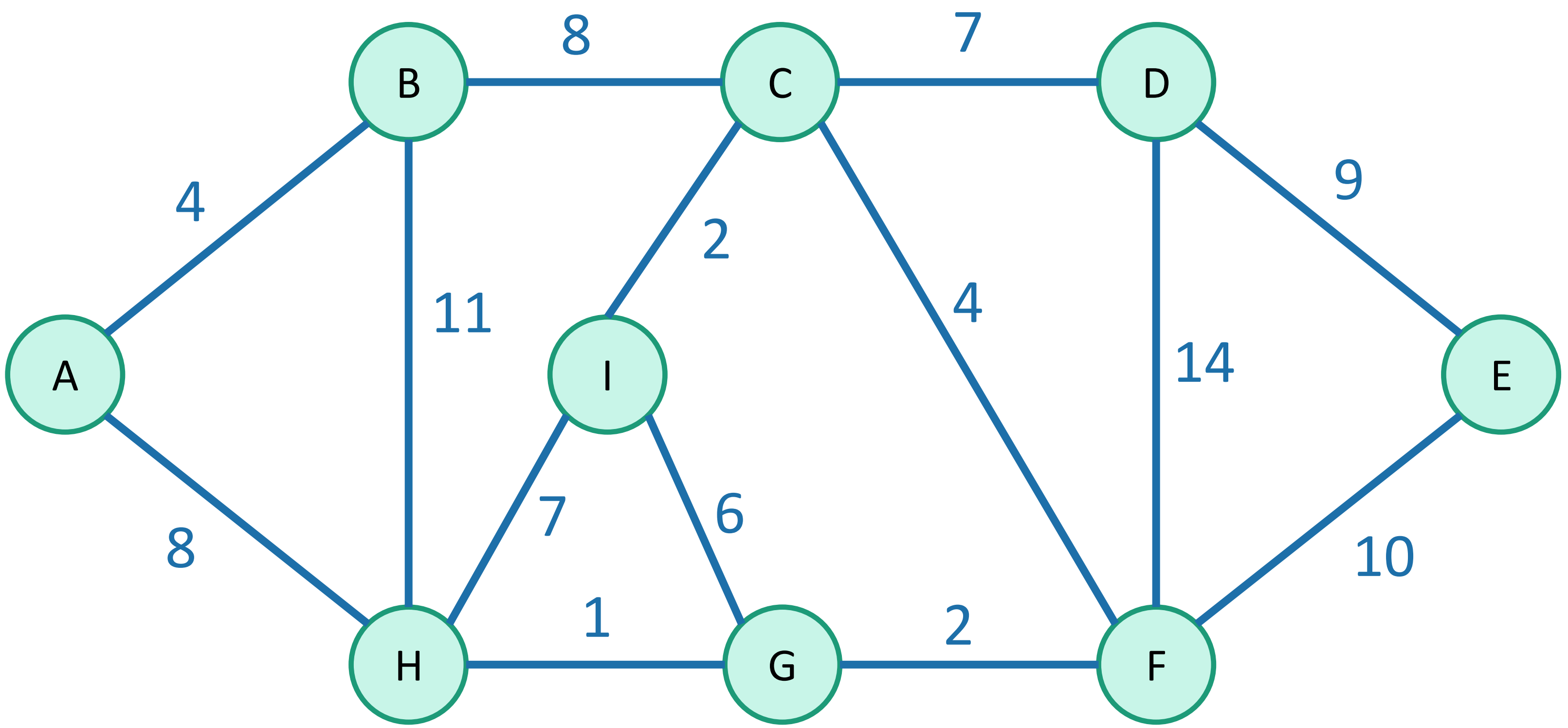
Kruskal Algorithm

Minimum Spanning Tree

That's not the only greedy algorithm

what if we just always take the cheapest edge?

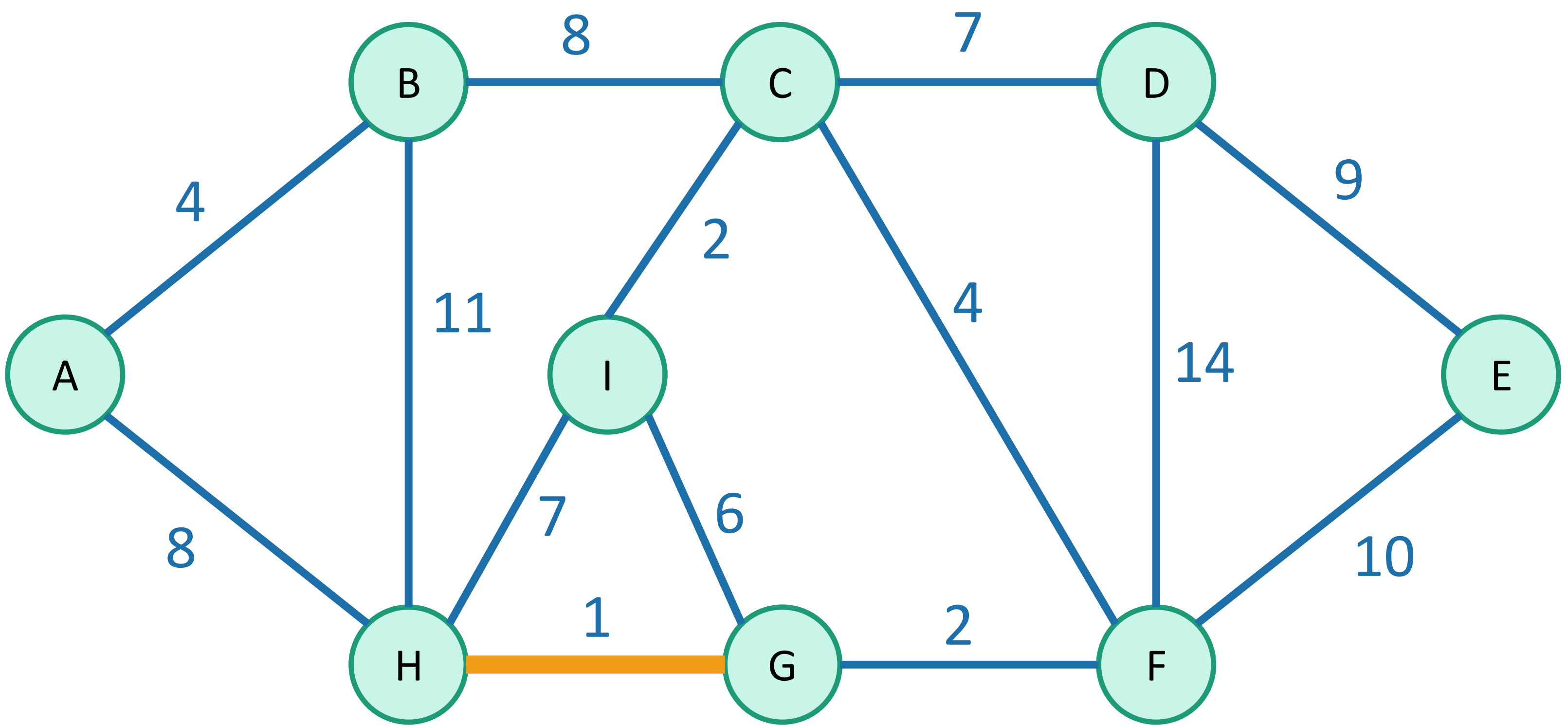
whether or not it's connected to what we have so far?



That’s not the only greedy algorithm

what if we just always take the cheapest edge?

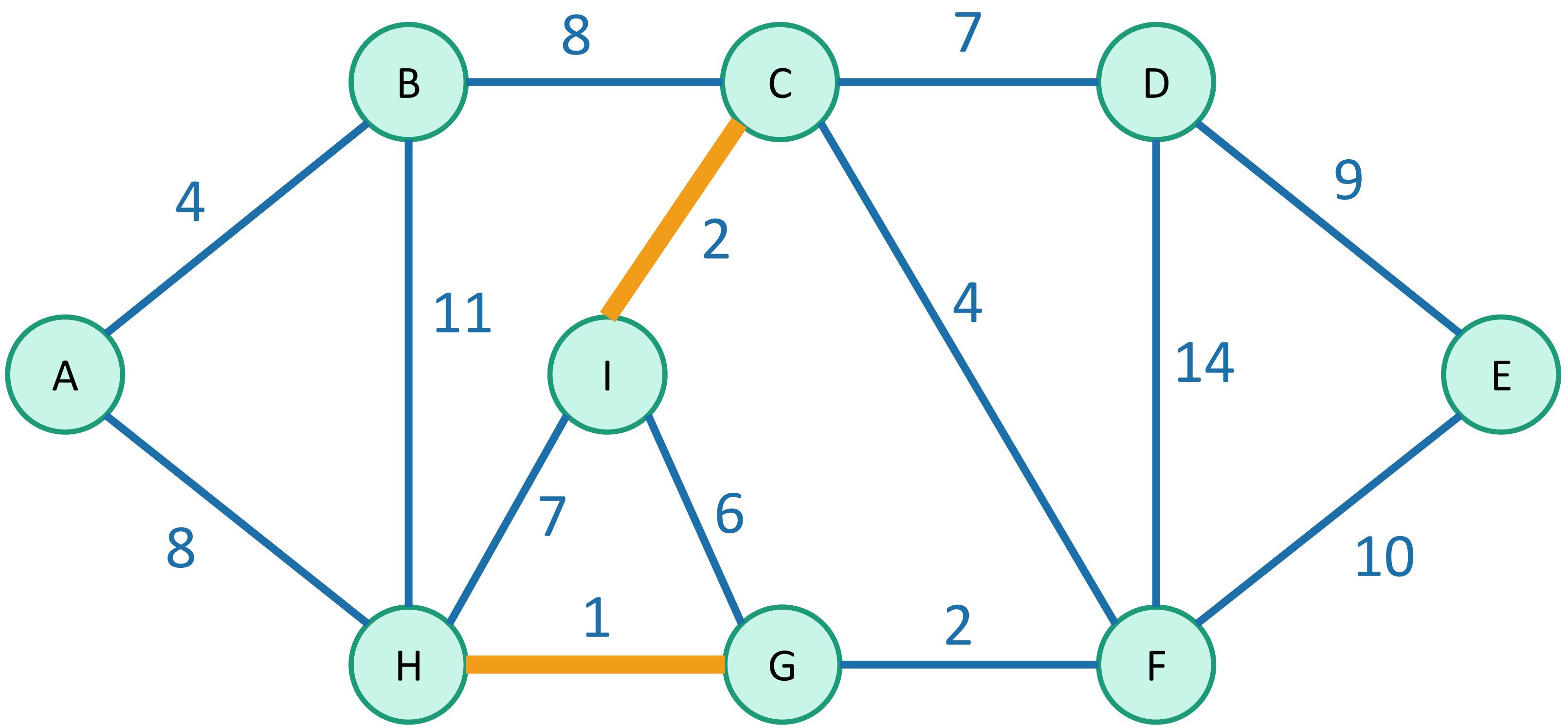
whether or not it’s connected to what we have so far?



That's not the only greedy algorithm

what if we just always take the cheapest edge?

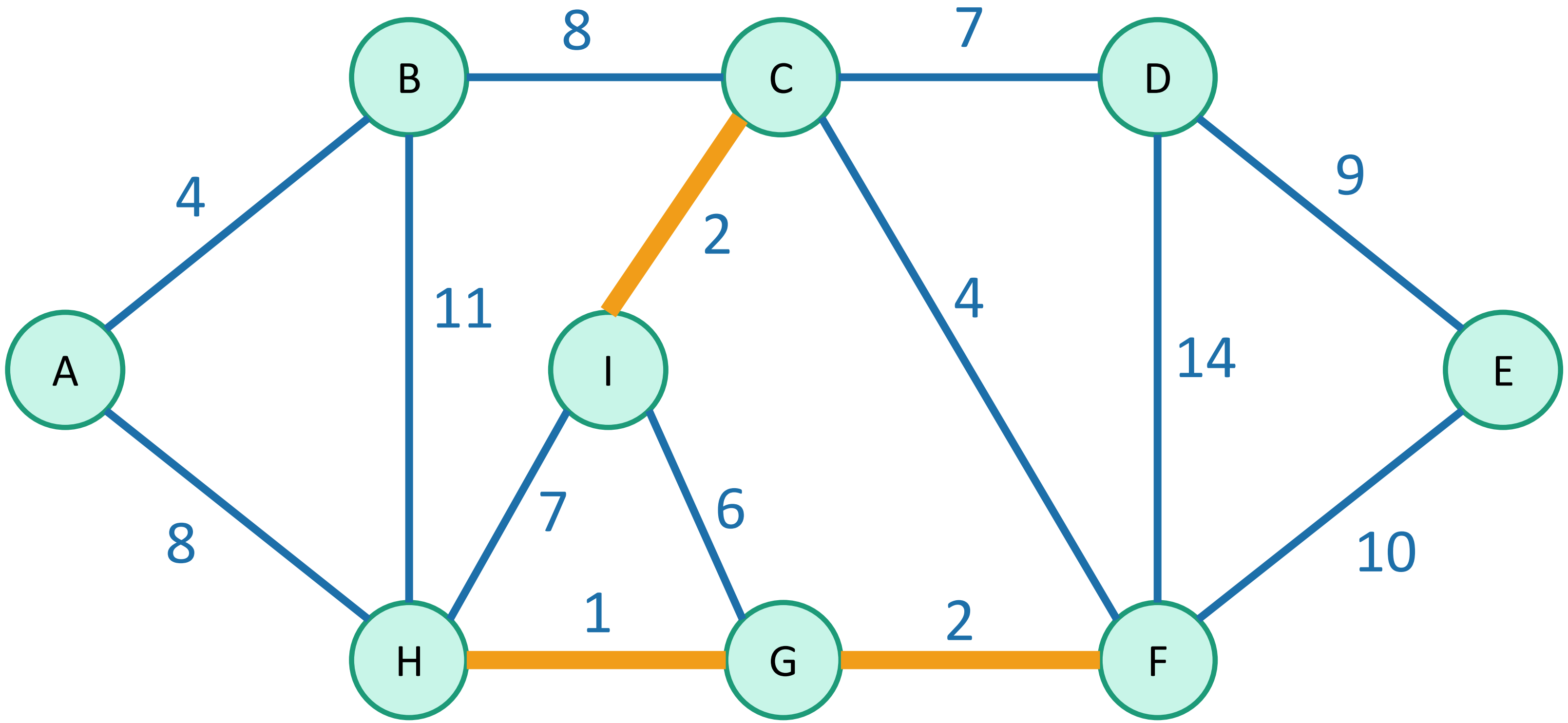
whether or not it's connected to what we have so far?



That's not the only greedy algorithm

what if we just always take the cheapest edge?

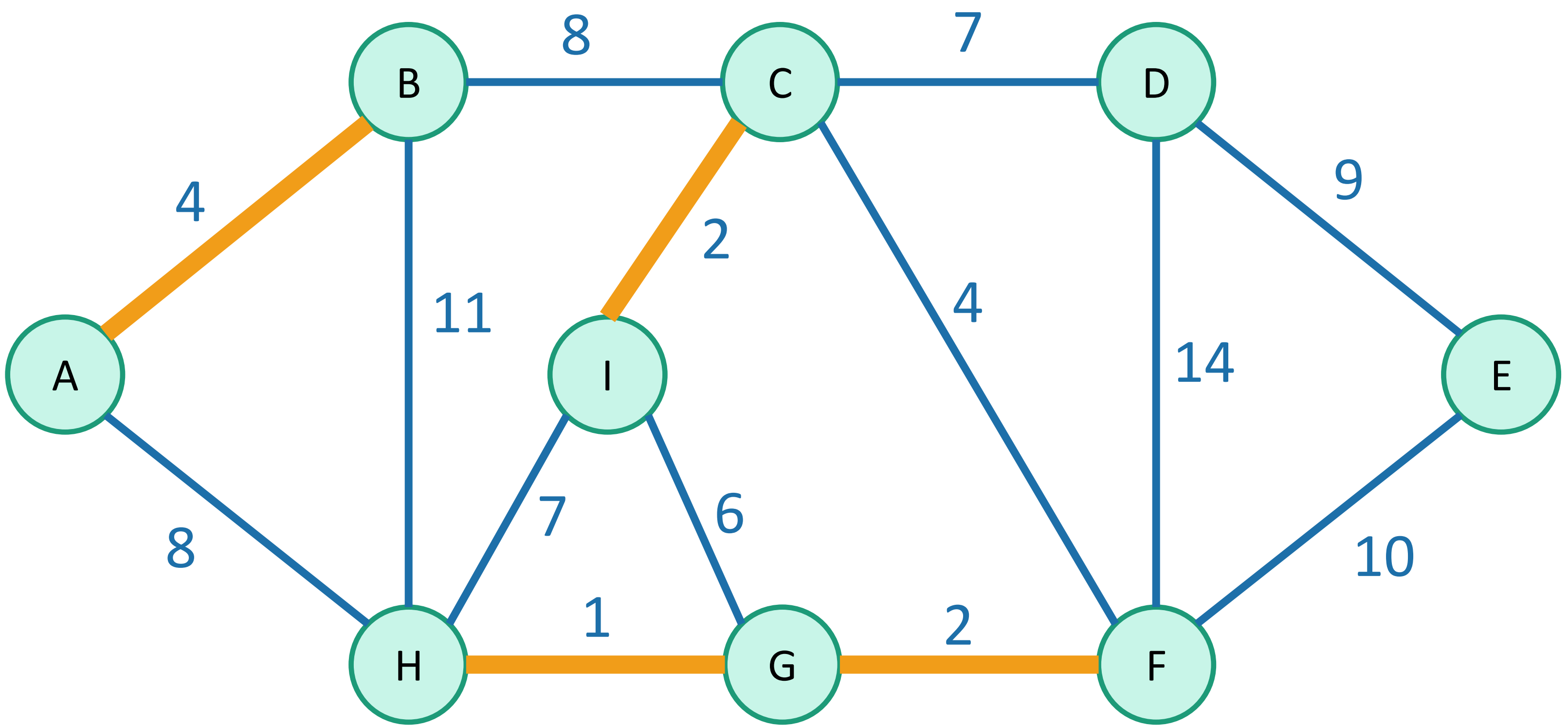
whether or not it's connected to what we have so far?



That’s not the only greedy algorithm

what if we just always take the cheapest edge?

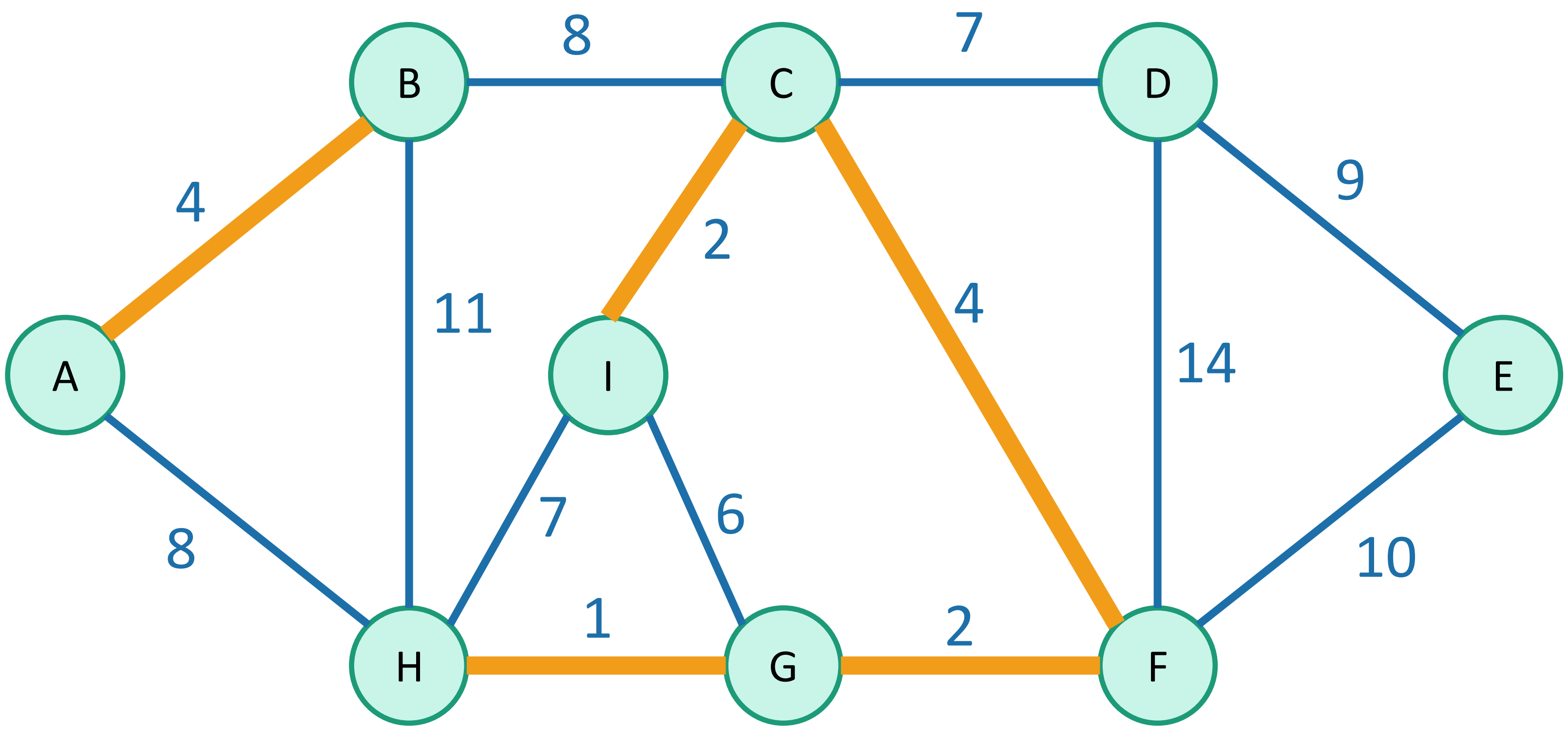
whether or not it’s connected to what we have so far?



That’s not the only greedy algorithm

what if we just always take the cheapest edge?

whether or not it’s connected to what we have so far?

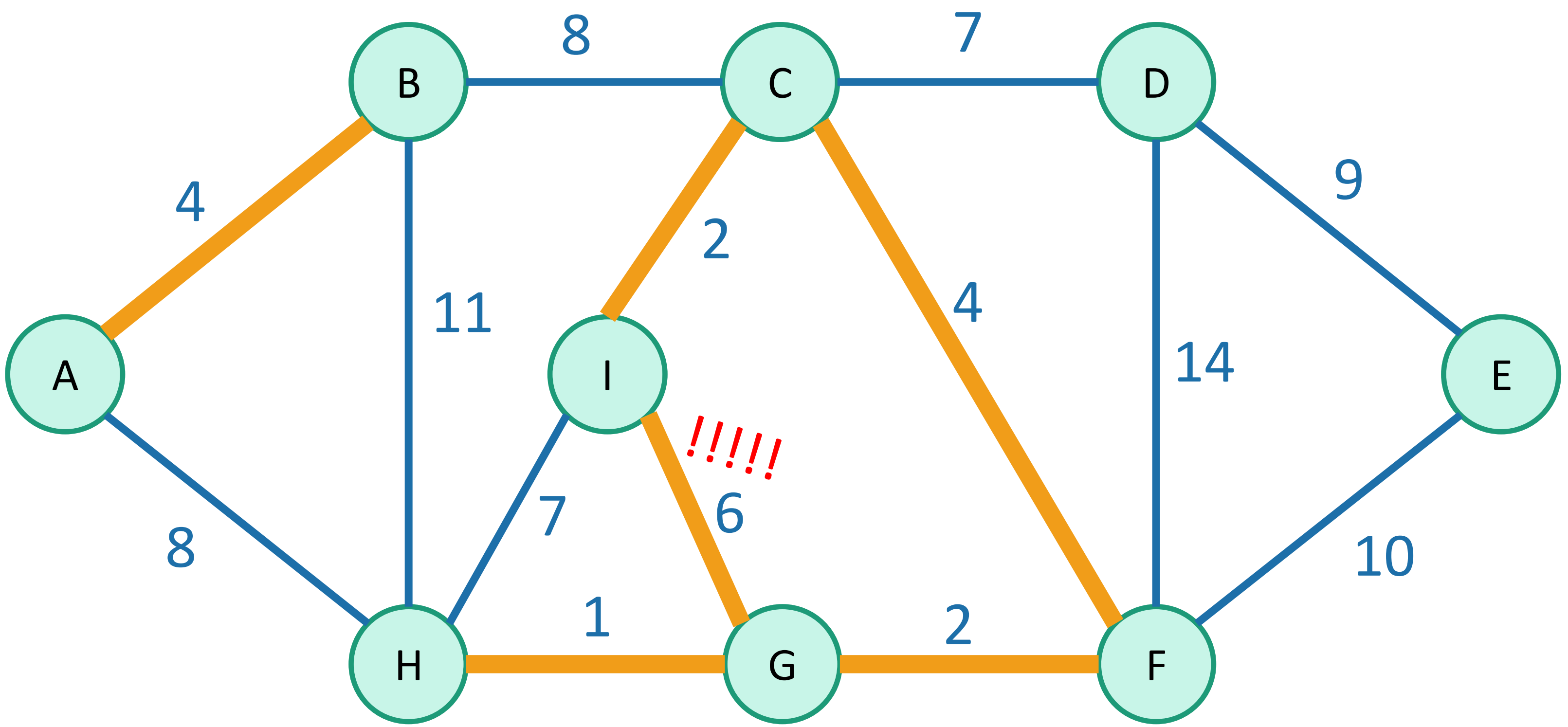


That's not the only greedy algorithm

what if we just always take the cheapest edge?

whether or not it's connected to what we have so far?

That won't cause a cycle

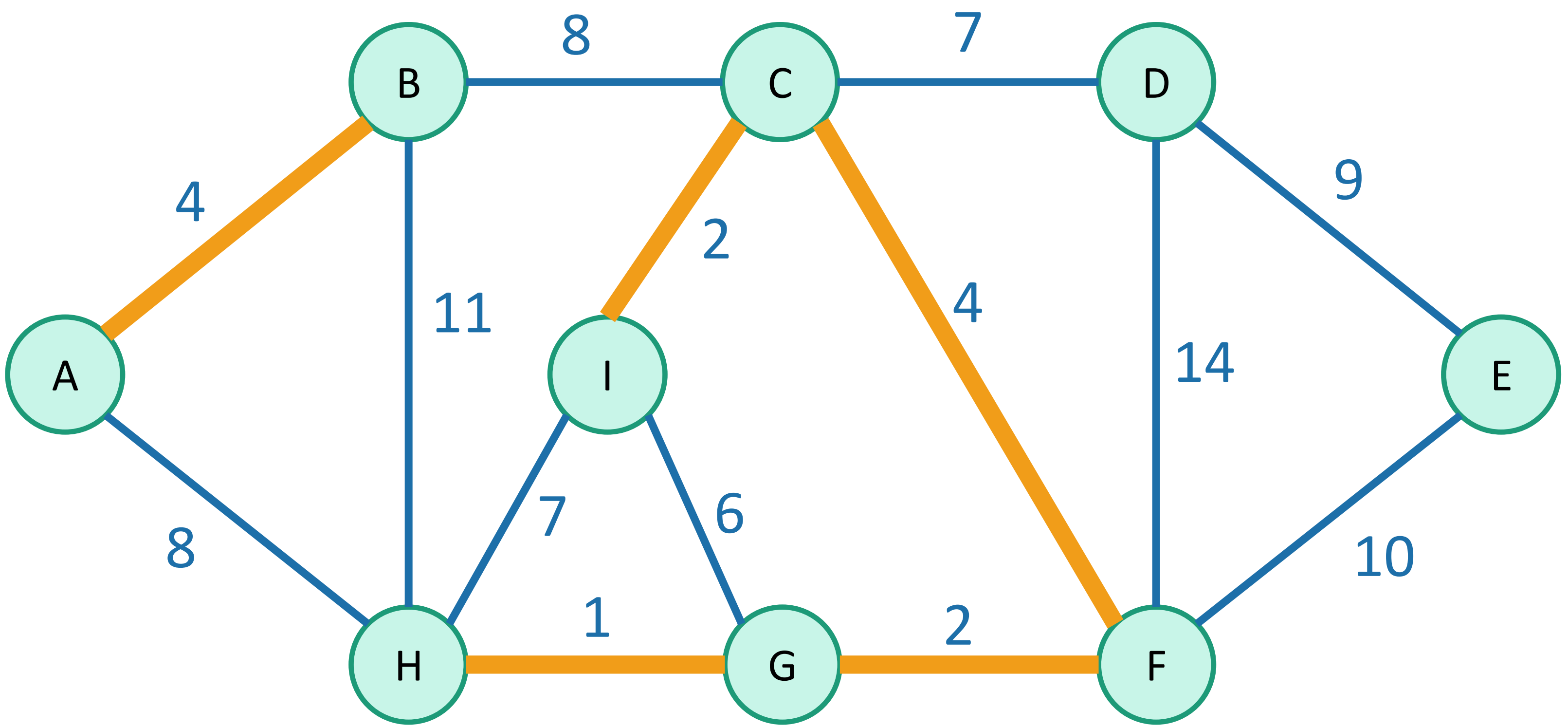


That's not the only greedy algorithm

what if we just always take the cheapest edge?

whether or not it's connected to what we have so far?

That won't cause a cycle

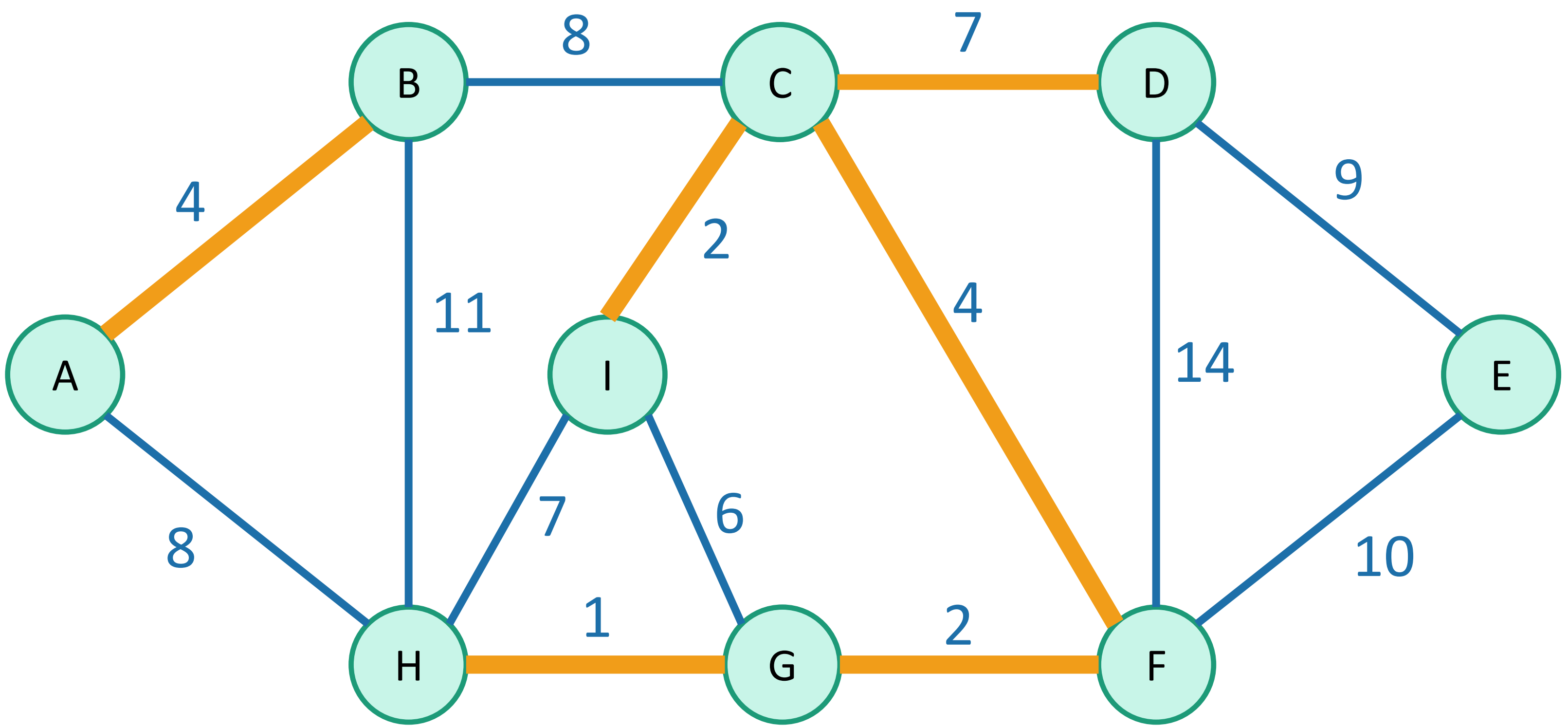


That's not the only greedy algorithm

what if we just always take the cheapest edge?

whether or not it's connected to what we have so far?

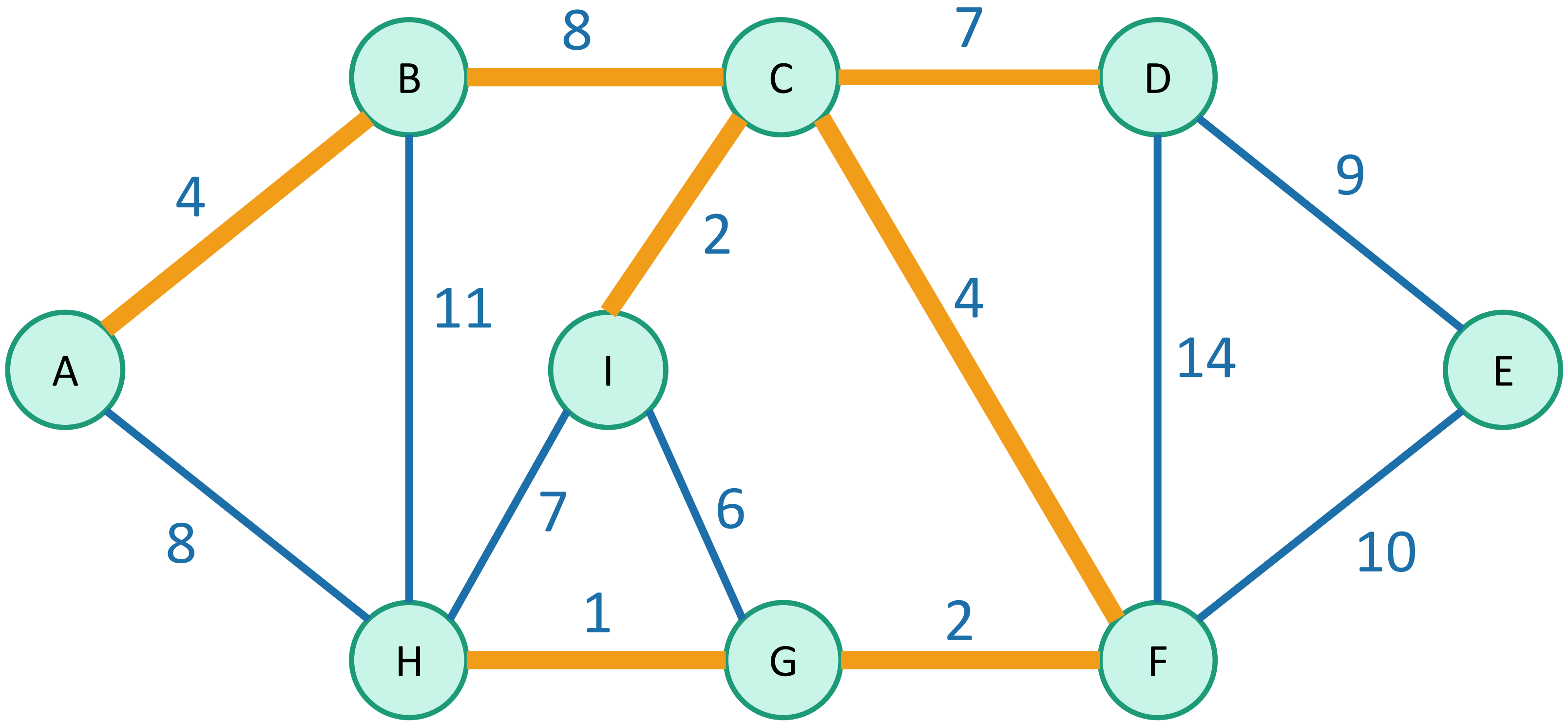
That won't cause a cycle



That's not the only greedy algorithm

what if we just always take the cheapest edge?
whether or not it's connected to what we have so far?

That won't cause a cycle

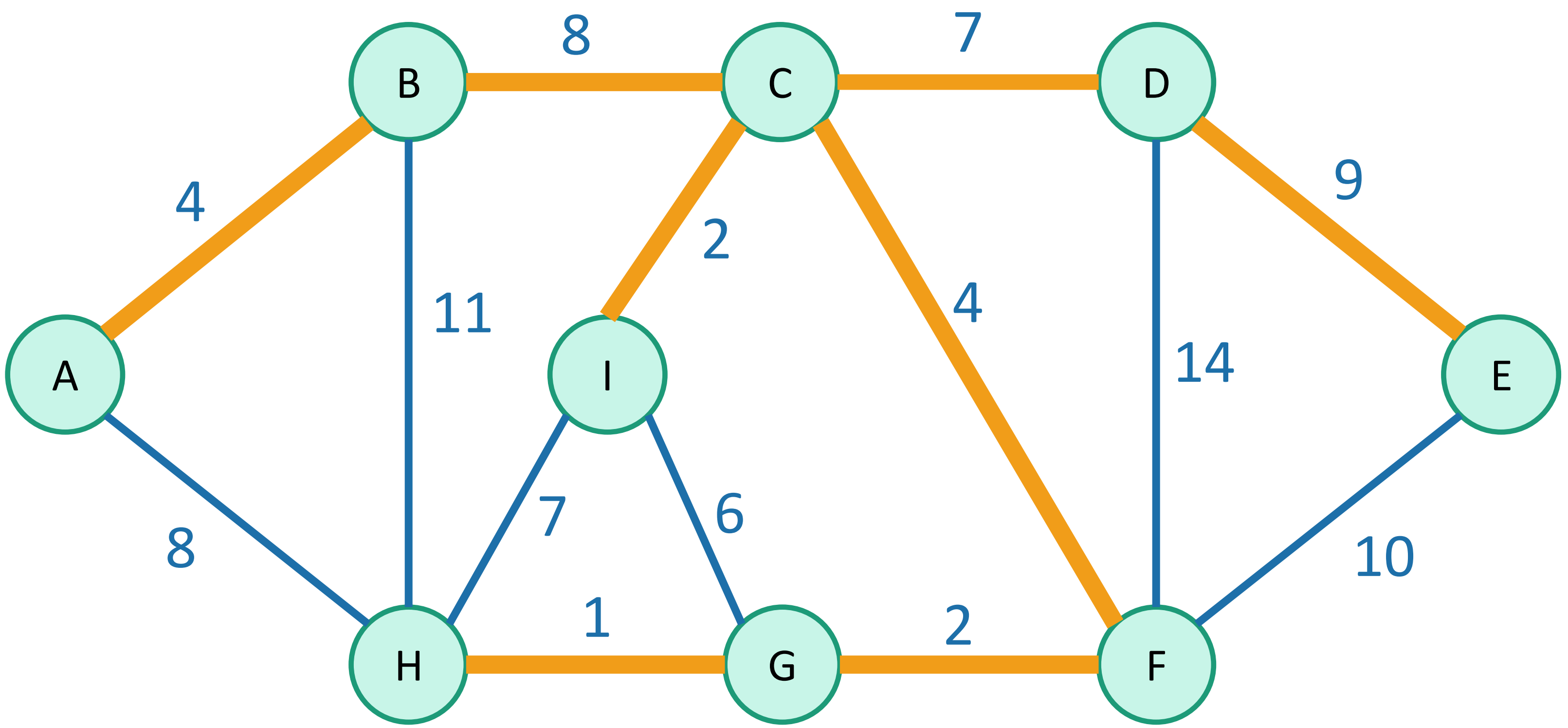


That’s not the only greedy algorithm

what if we just always take the cheapest edge?

whether or not it’s connected to what we have so far?

That won’t cause a cycle



We've discovered
Kruskal's algorithm!

- **slowKruskal**(G = (V,E)):
  - Sort the edges in E by non-decreasing weight.
  - MST = {}
  - **for** e in E (in sorted order):
    - **if** adding e to MST won't cause a cycle:
      - add e to MST.
  - **return** MST

m iterations through this loop

How do we check this?

How **would** you figure out if added e would make a cycle in this algorithm?

<u>Naively, the running time is ???:</u>

- For each of m iterations of the for loop:
  - Check if adding e would cause a cycle…

Two questions

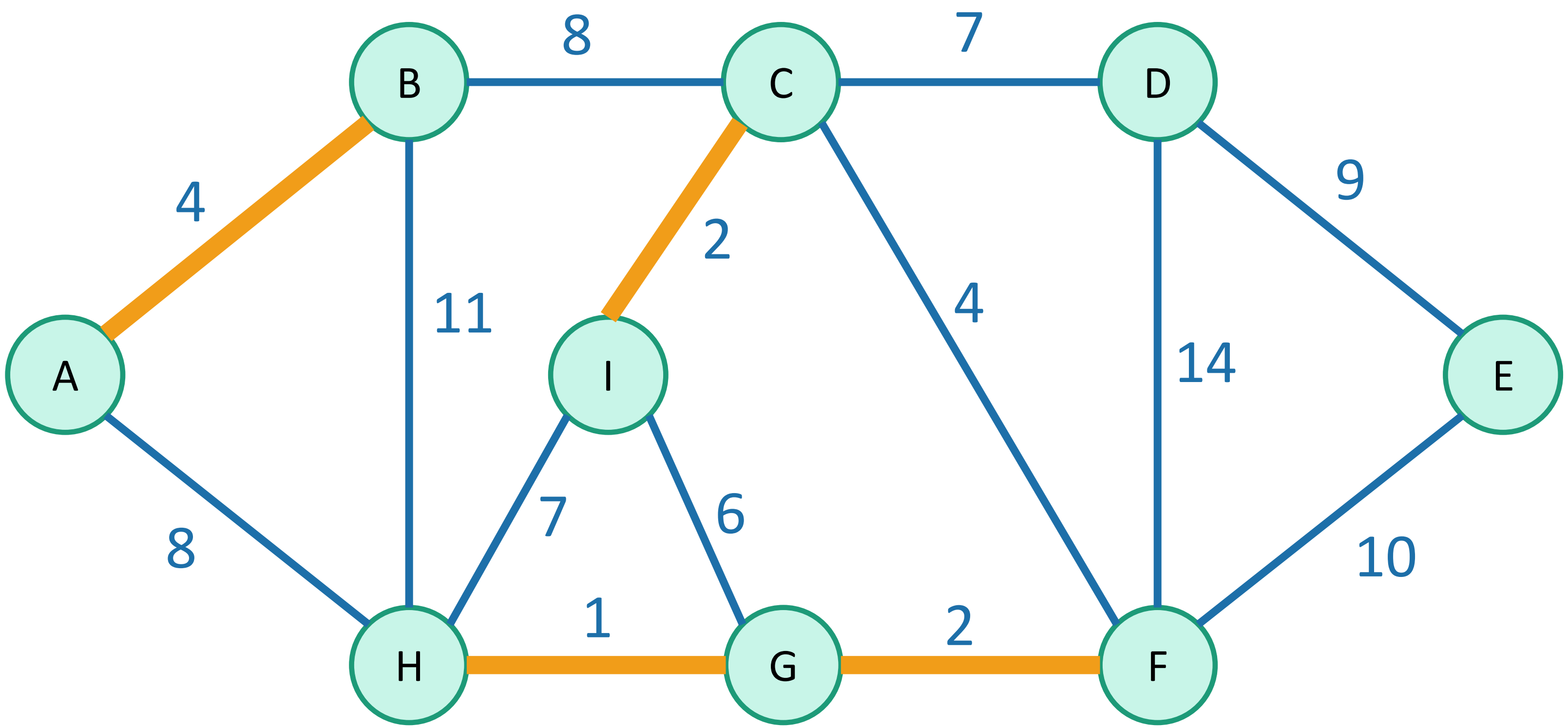
1. Does it work?
   - That is, does it actually return a MST?

2. How do we actually implement this?
   - the pseudocode above says “slowKruskal”…

Let’s do this one first

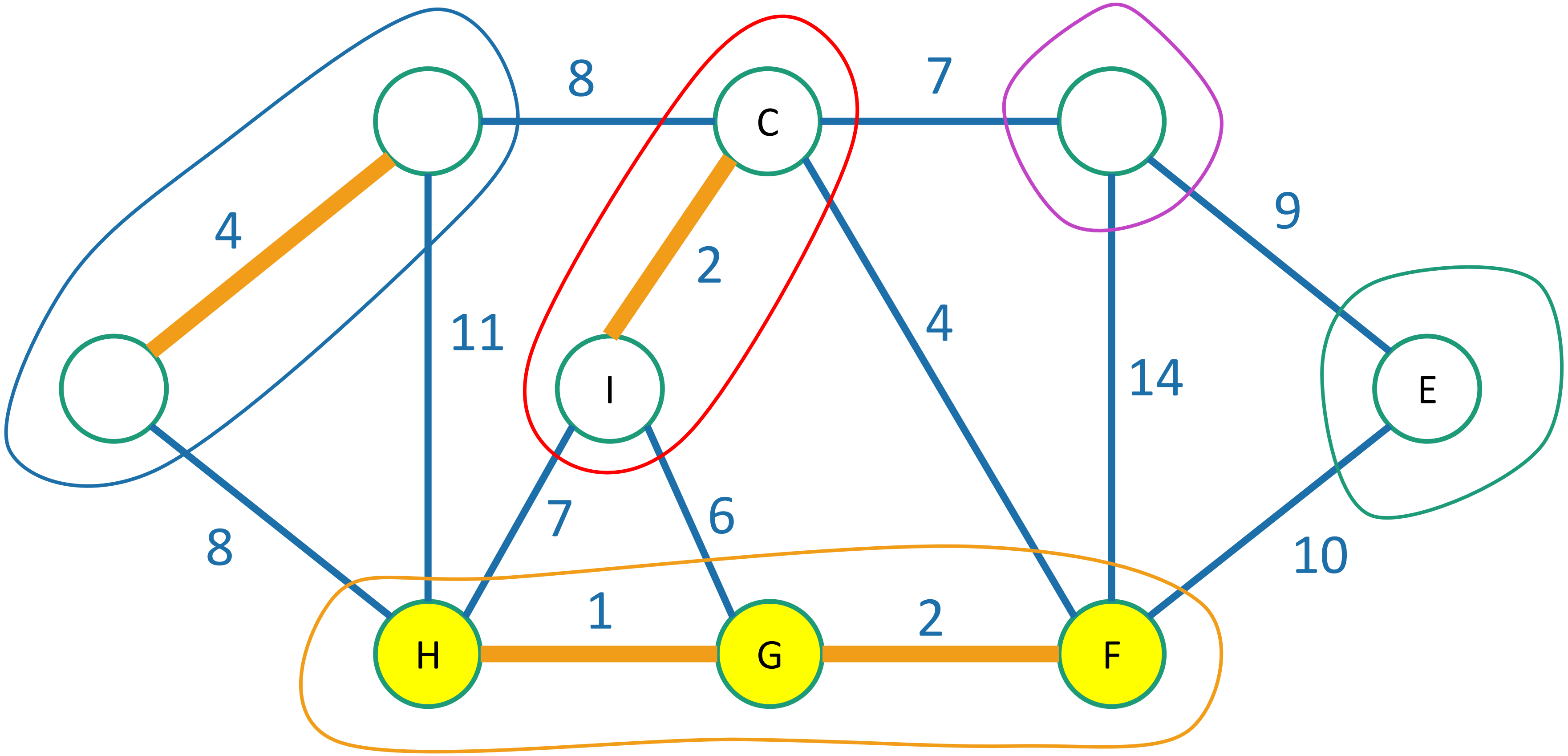
At each step of Kruskal's, we are maintaining a forest.

A **forest** is a collection of disjoint trees



At each step of Kruskal's, we are maintaining a forest.

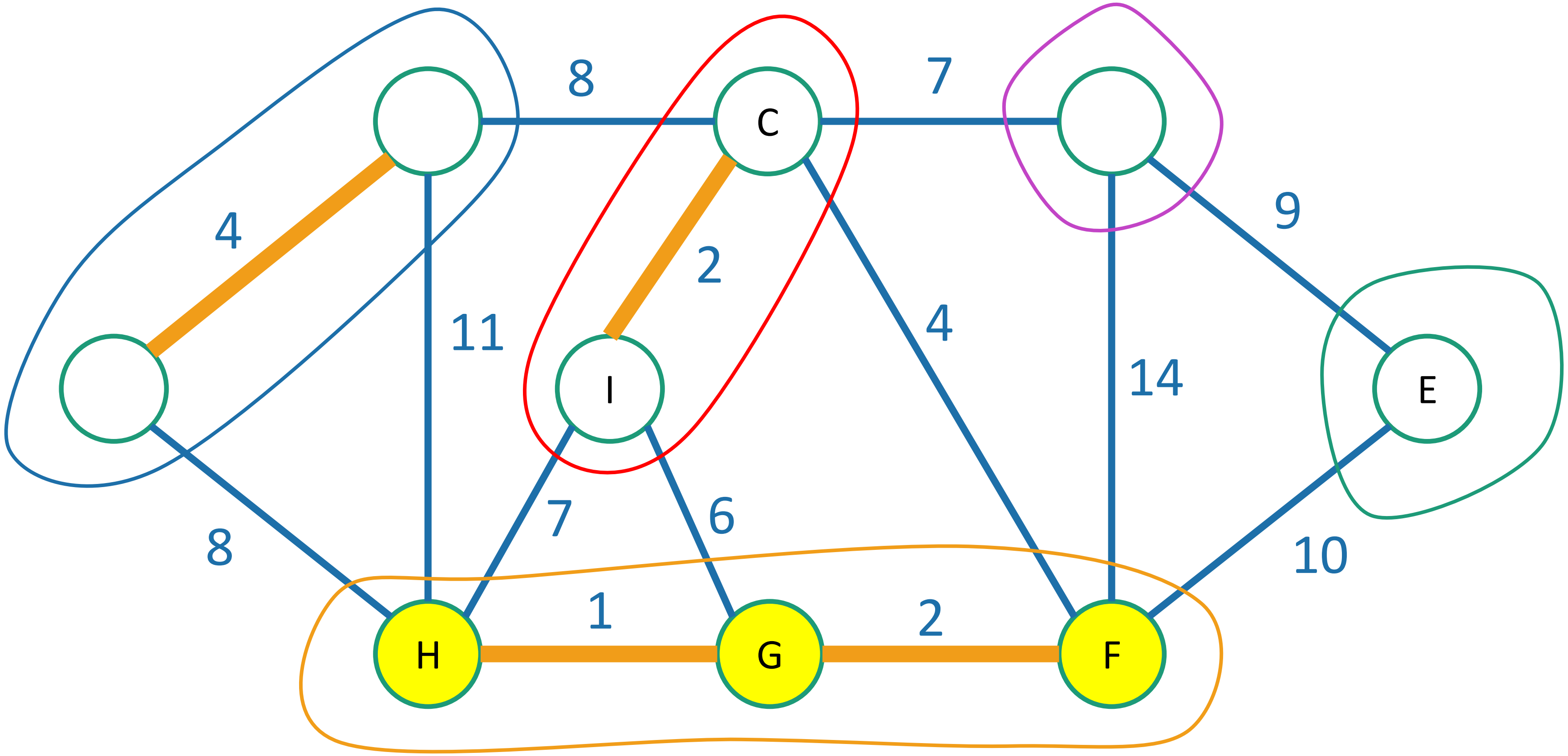
A **forest** is a collection of disjoint trees



At each step of Kruskal's, we are maintaining a forest.

A **forest** is a collection of disjoint trees

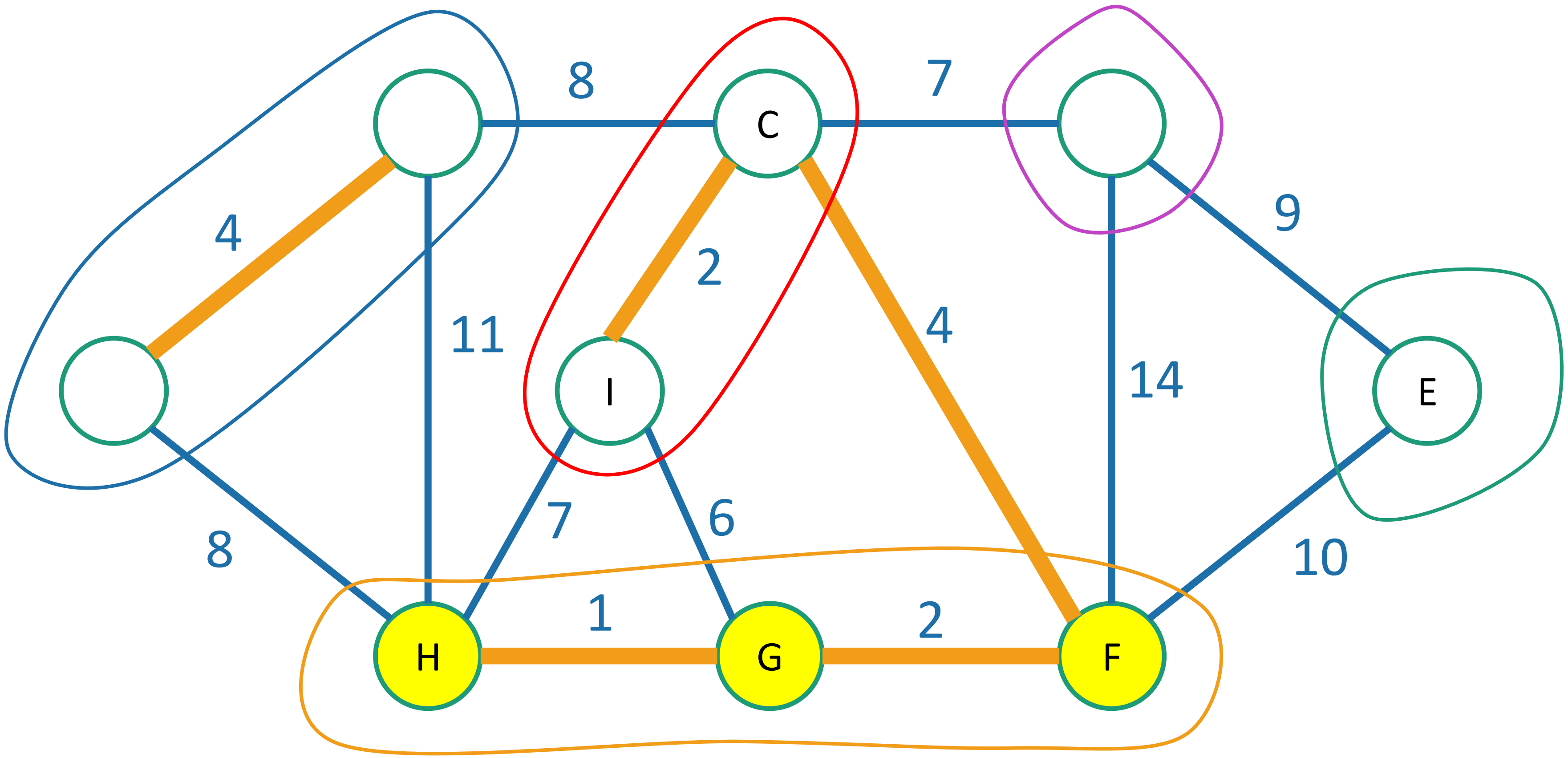
When we add an edge, we merge two trees:



At each step of Kruskal's, we are maintaining a forest.

A **forest** is a collection of disjoint trees

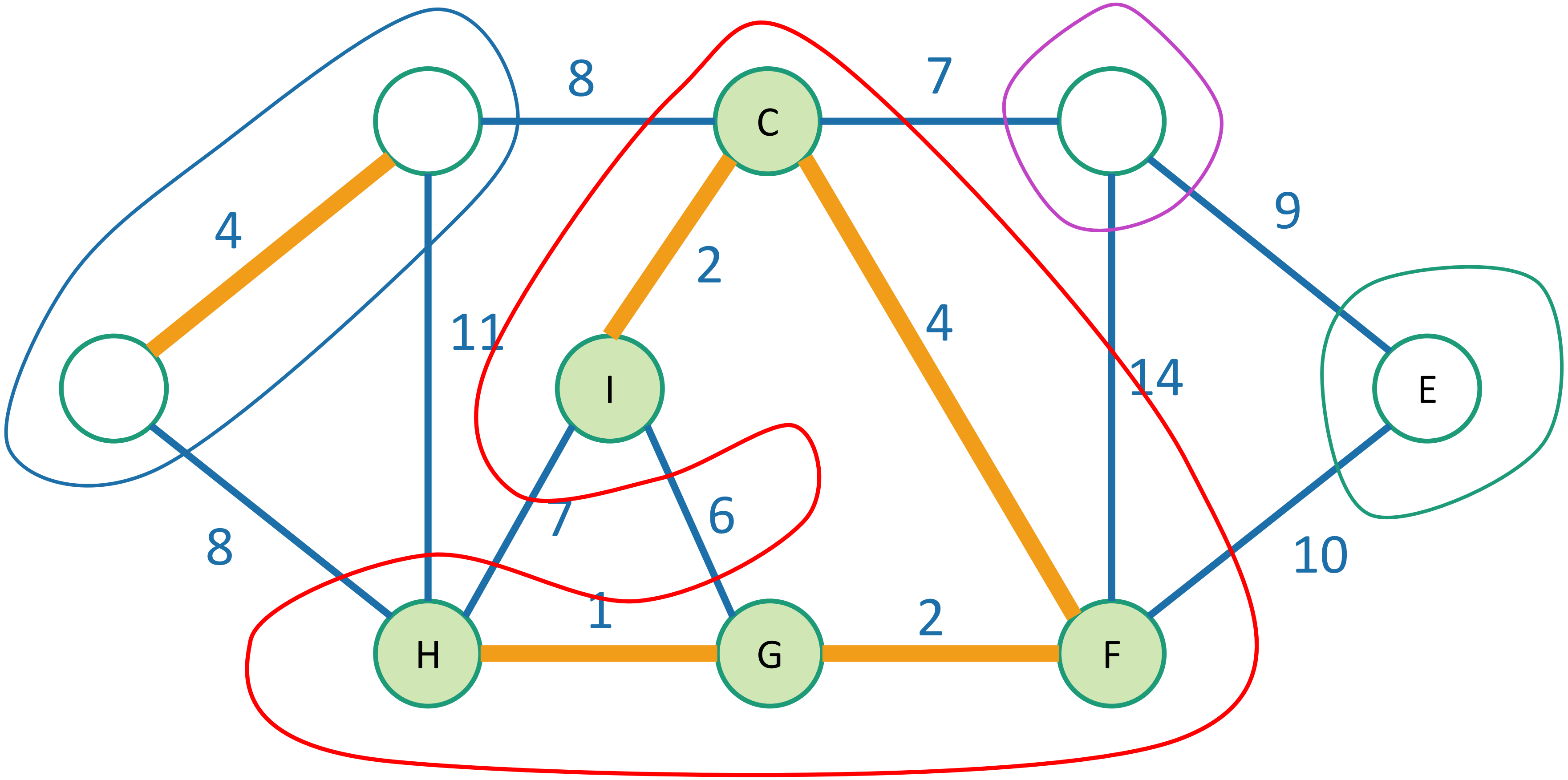
When we add an edge, we merge two trees:



At each step of Kruskal’s, we are maintaining a forest.

A **forest** is a collection of disjoint trees

When we add an edge, we merge two trees:



We never add an edge within a tree since that would create a cycle.

Keep the trees in a special data structure

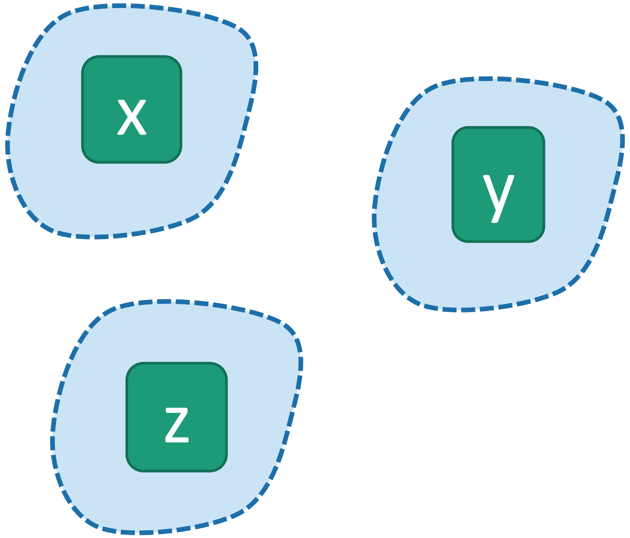
"treehouse"?

Union-find data structure
also called disjoint-set data structure

- Used for storing collections of sets
- Supports:
  - **makeSet(u):** create a set {u}
  - **find(u):** return the set that u is in
  - **union(u,v):** merge the set that u is in with the set that v is in.

```
makeSet(x)
makeSet(y)
makeSet(z)

union(x,y)
```

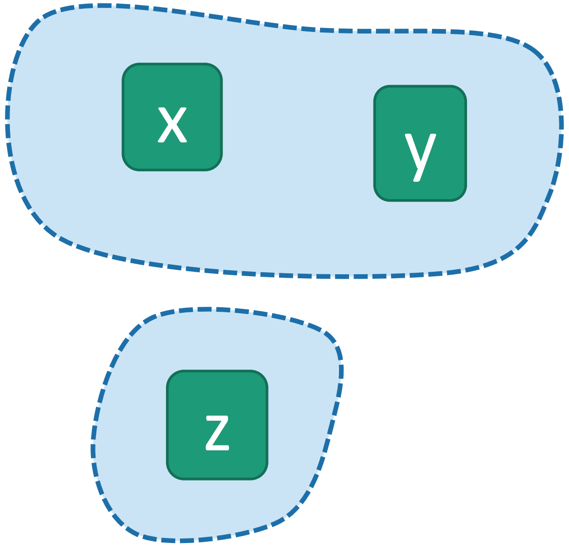


Union-find data structure
also called disjoint-set data structure

- Used for storing collections of sets
- Supports:
  - **makeSet(u):** create a set {u}
  - **find(u):** return the set that u is in
  - **union(u,v):** merge the set that u is in with the set that v is in.

```
makeSet(x)
makeSet(y)
makeSet(z)

union(x,y)
```



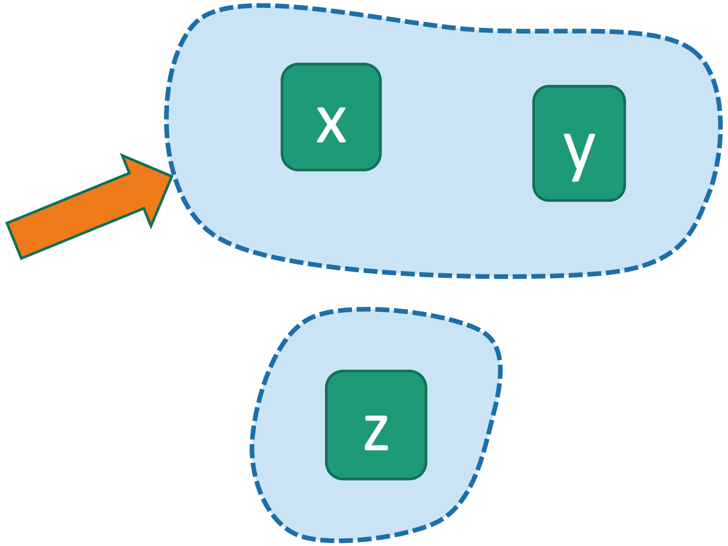
Union-find data structure
also called disjoint-set data structure

- Used for storing collections of sets
- Supports:
  - **makeSet(u):** create a set {u}
  - **find(u):** return the set that u is in
  - **union(u,v):** merge the set that u is in with the set that v is in.

```
makeSet(x)
makeSet(y)
makeSet(z)

union(x,y)

find(x)
```

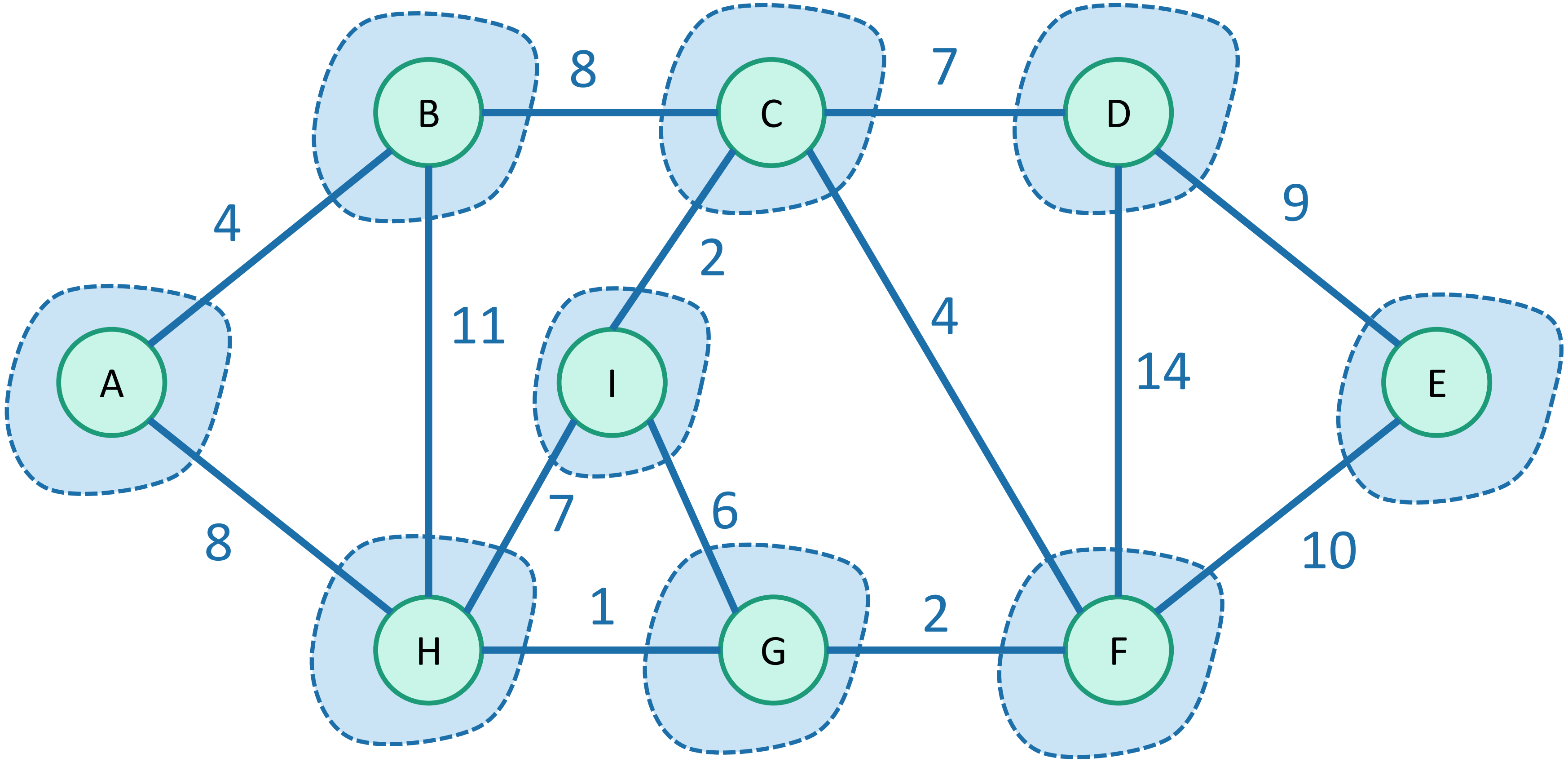


Kruskal pseudo-code

- **kruskal**(G = (V,E)):
  - Sort E by weight in non-decreasing order
  - MST = {} // initialize an empty tree
  - **for** v in V:
    - **makeSet**(v) // put each vertex in its own tree in the forest
  - **for** (u,v) in E: // go through the edges in sorted order
    - **if find**(u) != **find**(v): // if u and v are not in the same tree
      - add (u,v) to MST
      - **union**(u,v) // merge u's tree with v's tree
  - **return** MST

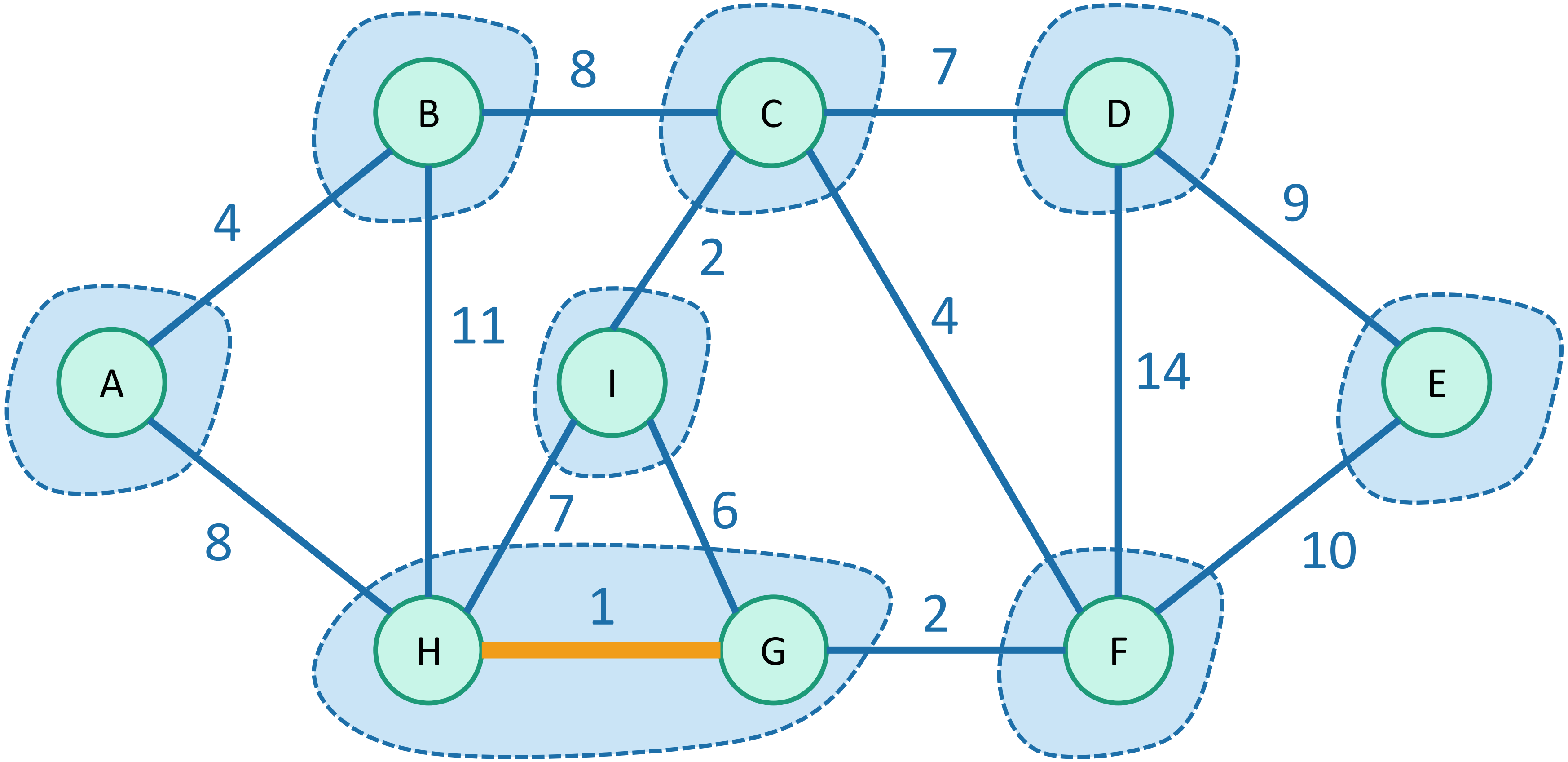
Once more…

To start, every vertex is in its own tree.



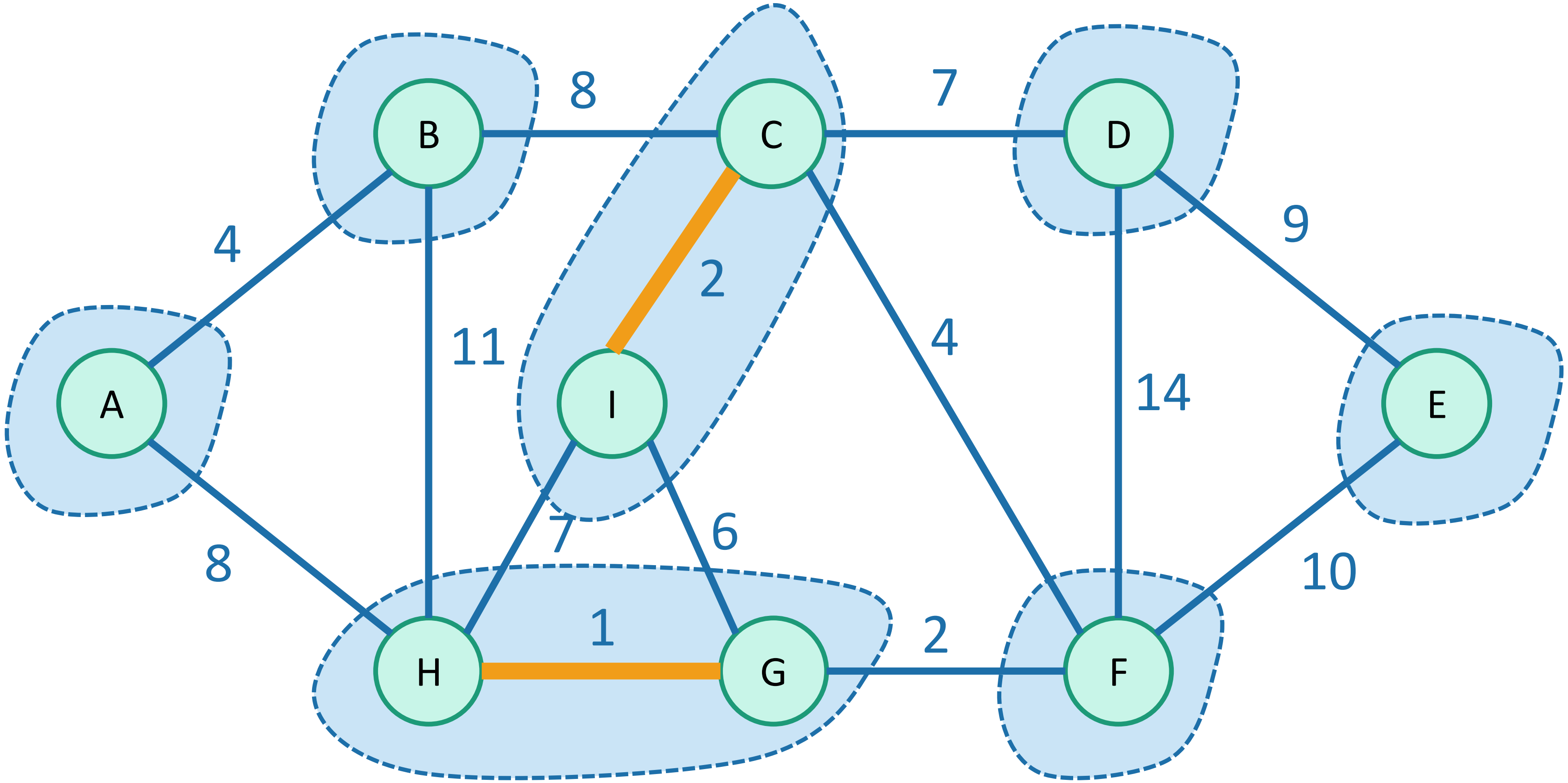
Once more…

Then start merging.



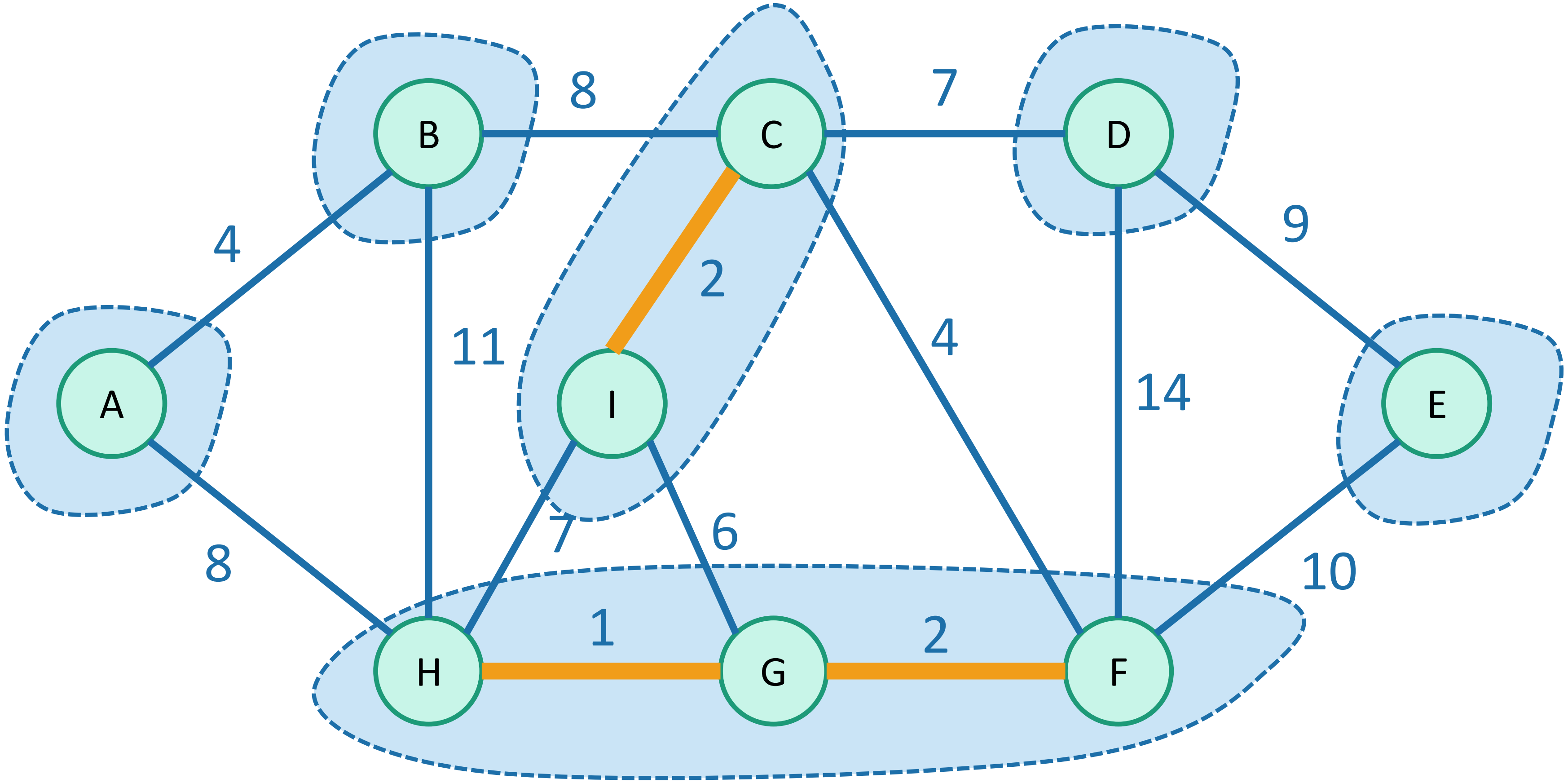
Once more…

Then start merging.



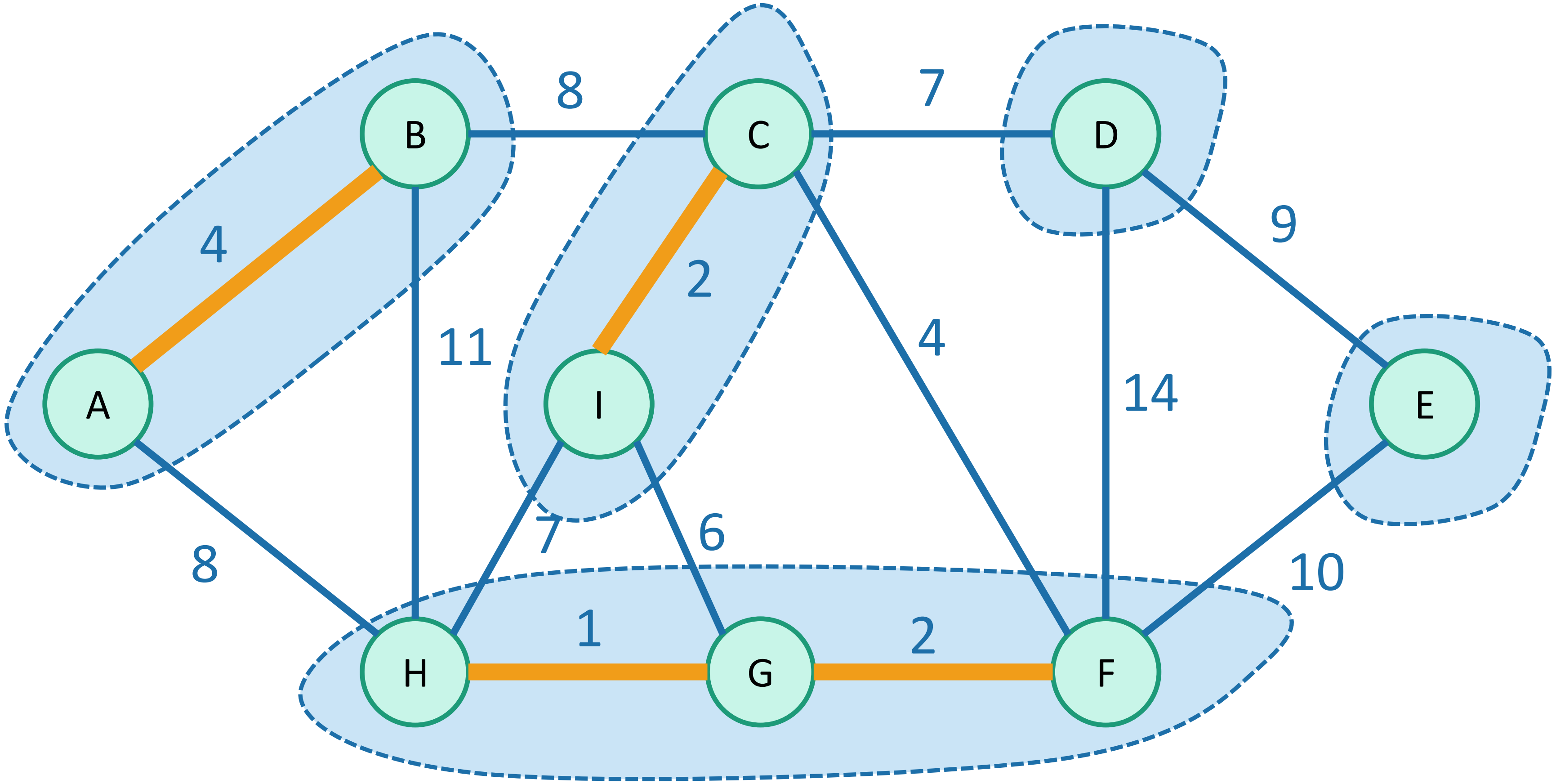
Once more…

Then start merging.



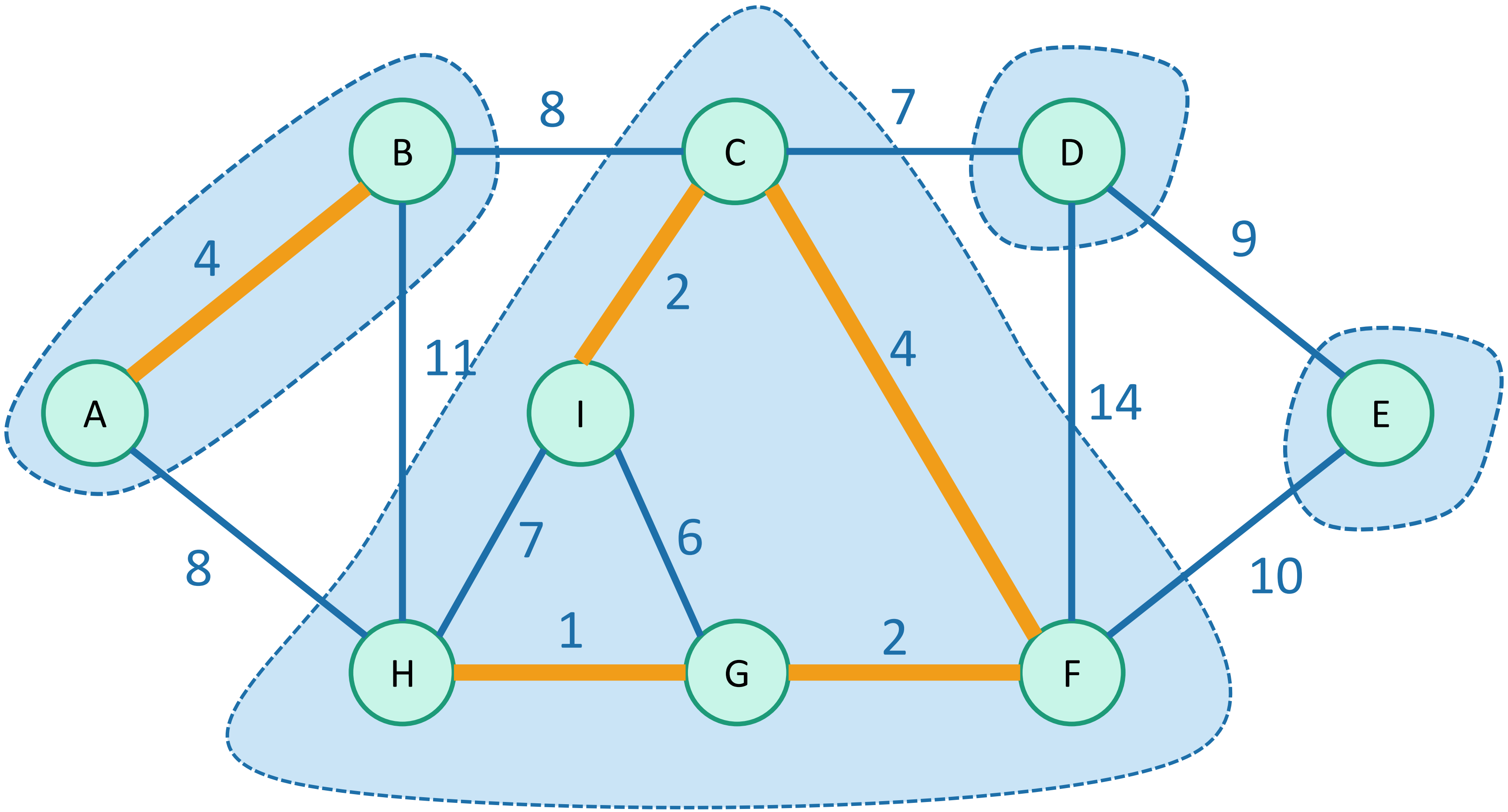
Once more…

Then start merging.



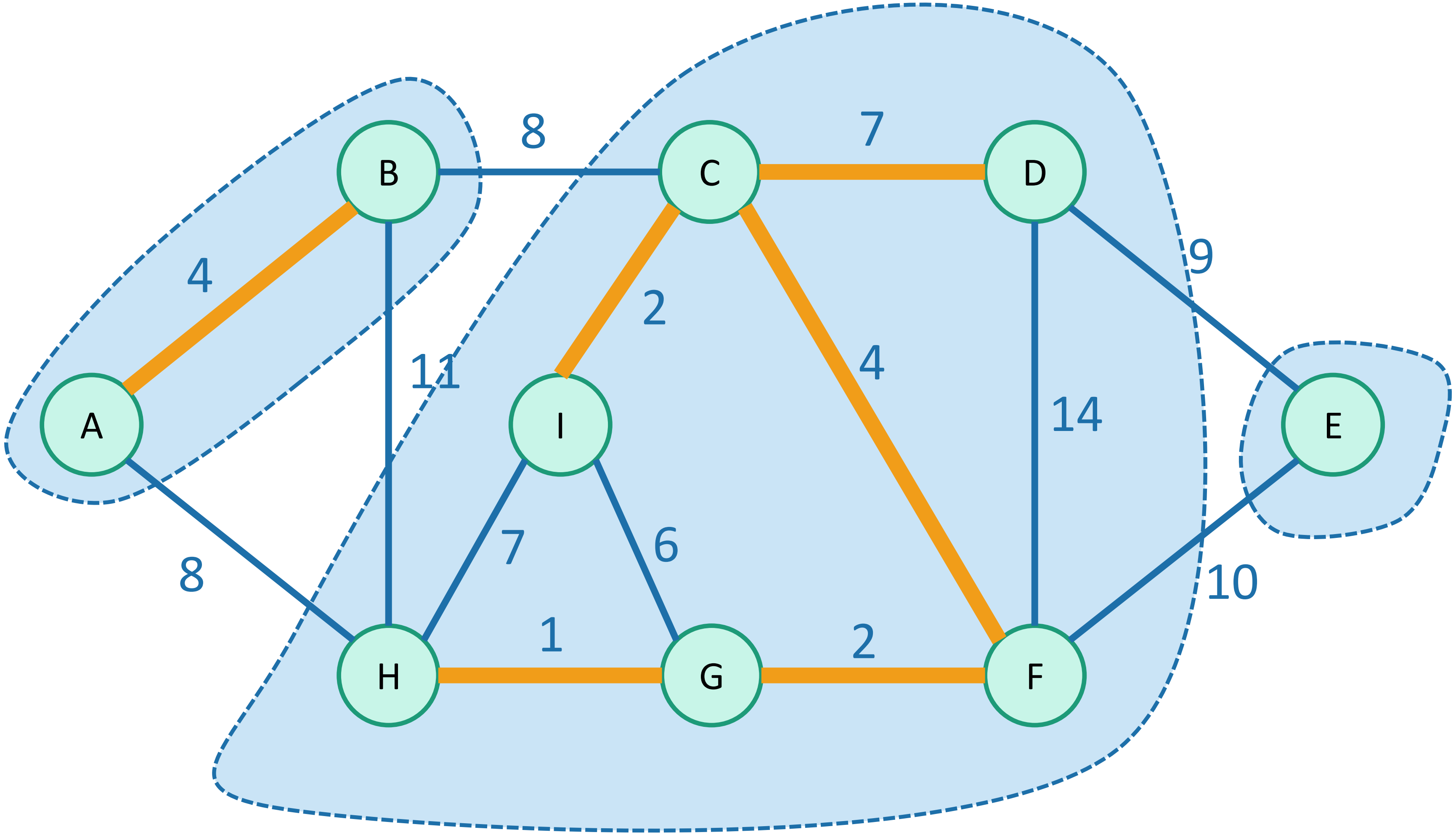
Once more…

Then start merging.



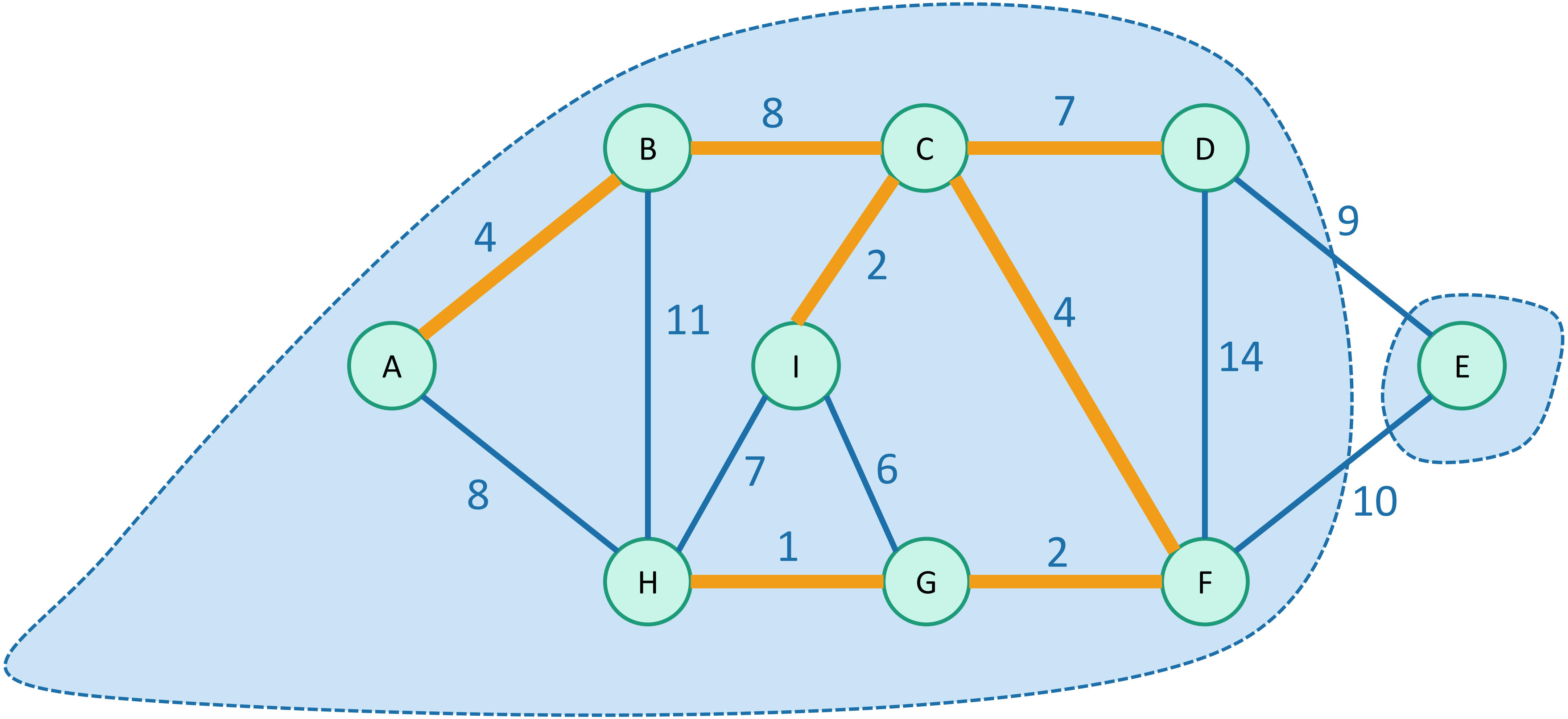
Once more…

Then start merging.



Once more…

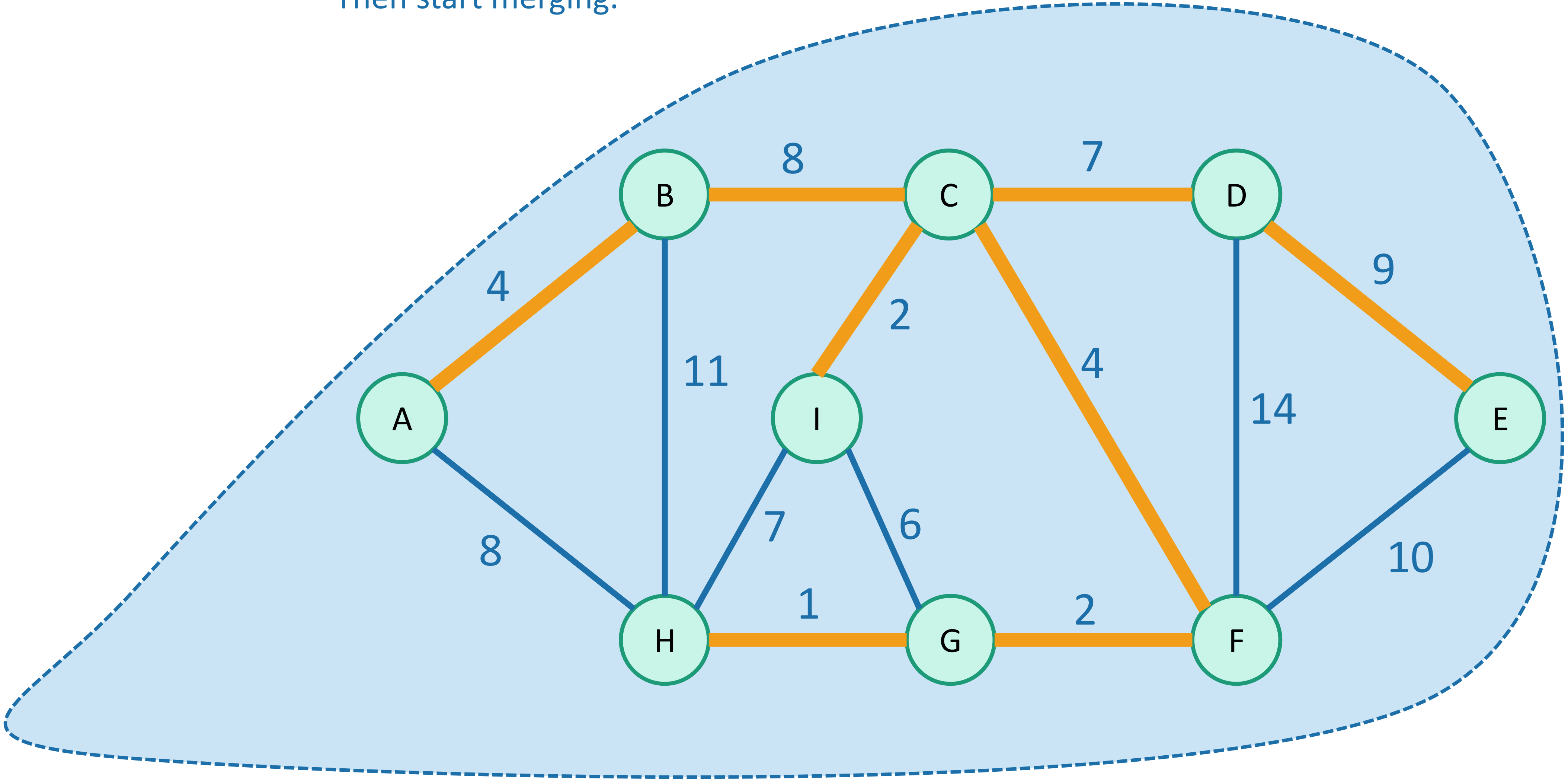
Then start merging.



Stop when we have one big tree!

Once more…

Then start merging.



Running time

- Sorting the edges takes O(m log(n))
  - In practice, if the weights are small integers we can use radixSort and take time O(m)
- For the rest:
  - n calls to **makeSet**
    - put each vertex in its own set
  - 2m calls to **find**
    - for each edge, **find** its endpoints
  - n calls to **union**
    - we will never add more than n-1 edges to the tree,
    - so we will never call **union** more than n-1 times.
- Total running time:
  - Worst-case O(mlog(n)), just like Prim.
  - Closer to O(m) if you can do radixSort

In practice, each of makeSet, find, and union run in constant time*

Two questions

1. Does it work?
   - That is, does it actually return a MST?

Now that we understand this "tree-merging" view, let's do this one.

2. How do we actually implement this?
   - the pseudocode above says "slowKruskal"…
     - **Worst-case running time O(mlog(n)) using a union-find data structure.**

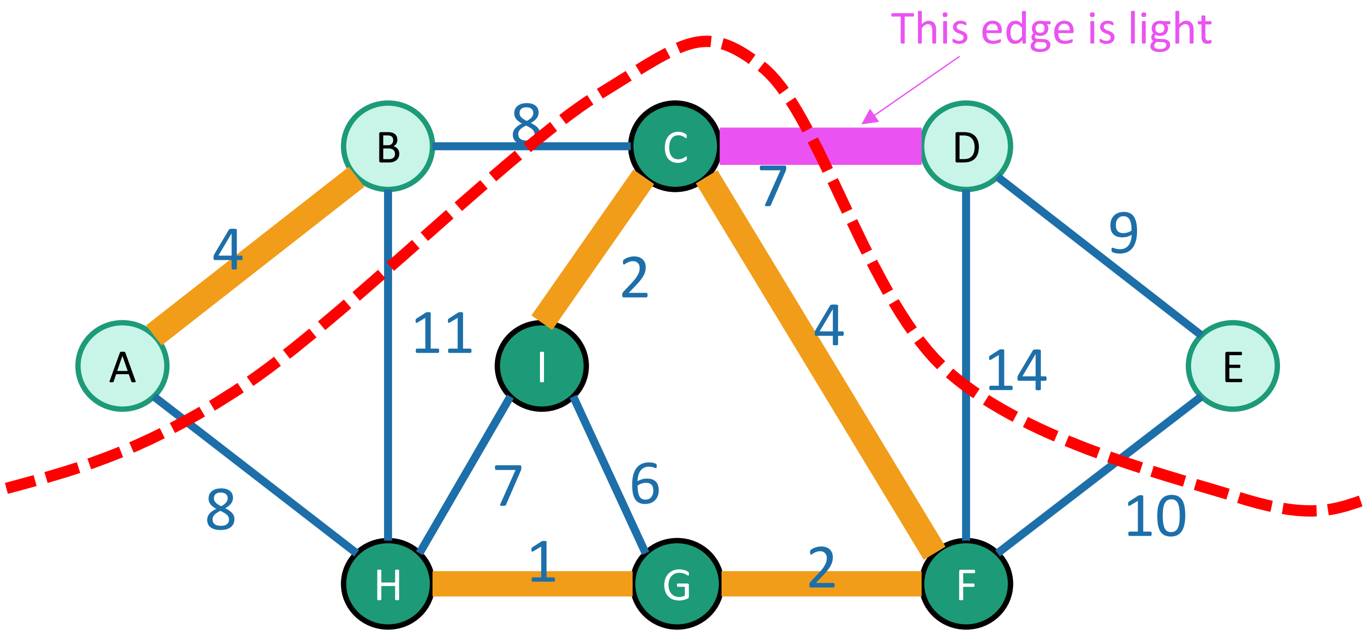
Does it work?

- We need to show that our greedy choices **don't rule out success.**
- That is, at every step:
  - There exists an MST that contains all of the edges we have added so far.
- Now it is time to use our lemma!

again!

Lemma

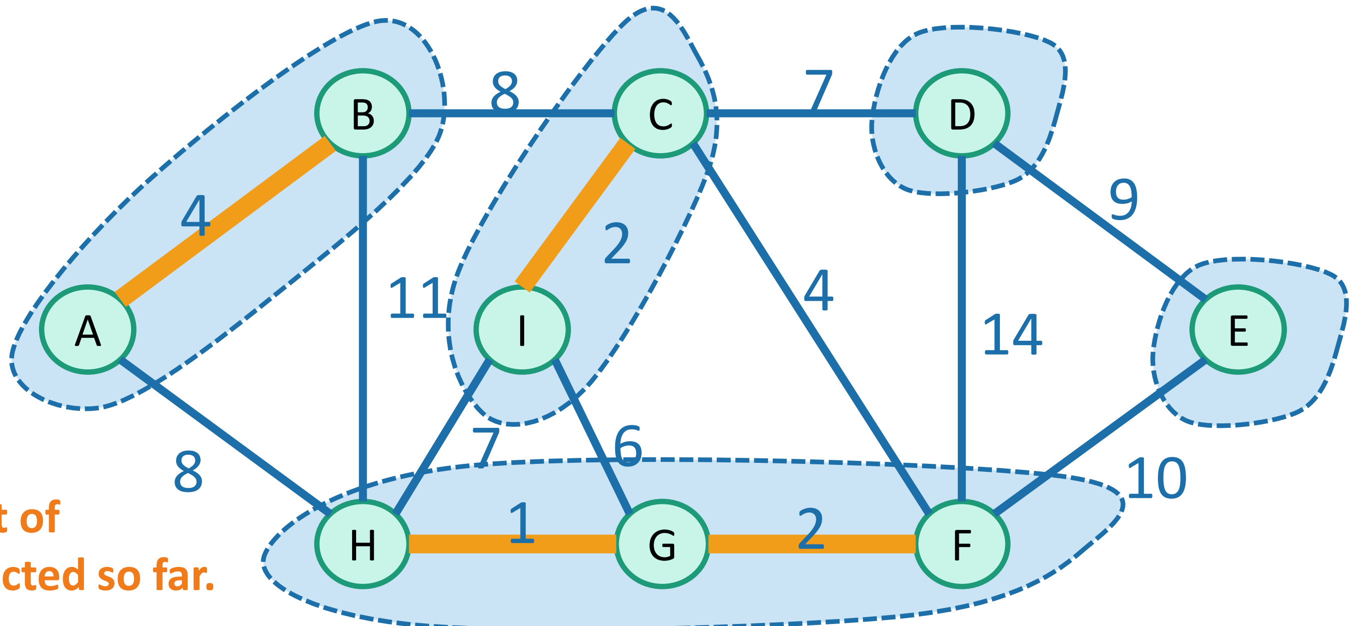
- Let S be a set of edges, and consider a cut that respects S.
- Suppose there is an MST containing S.
- Let (u,v) be a light edge.
- Then there is an MST containing S ∪ {(u,v)}



S is the set of **thick orange** edges

Partway through Kruskal

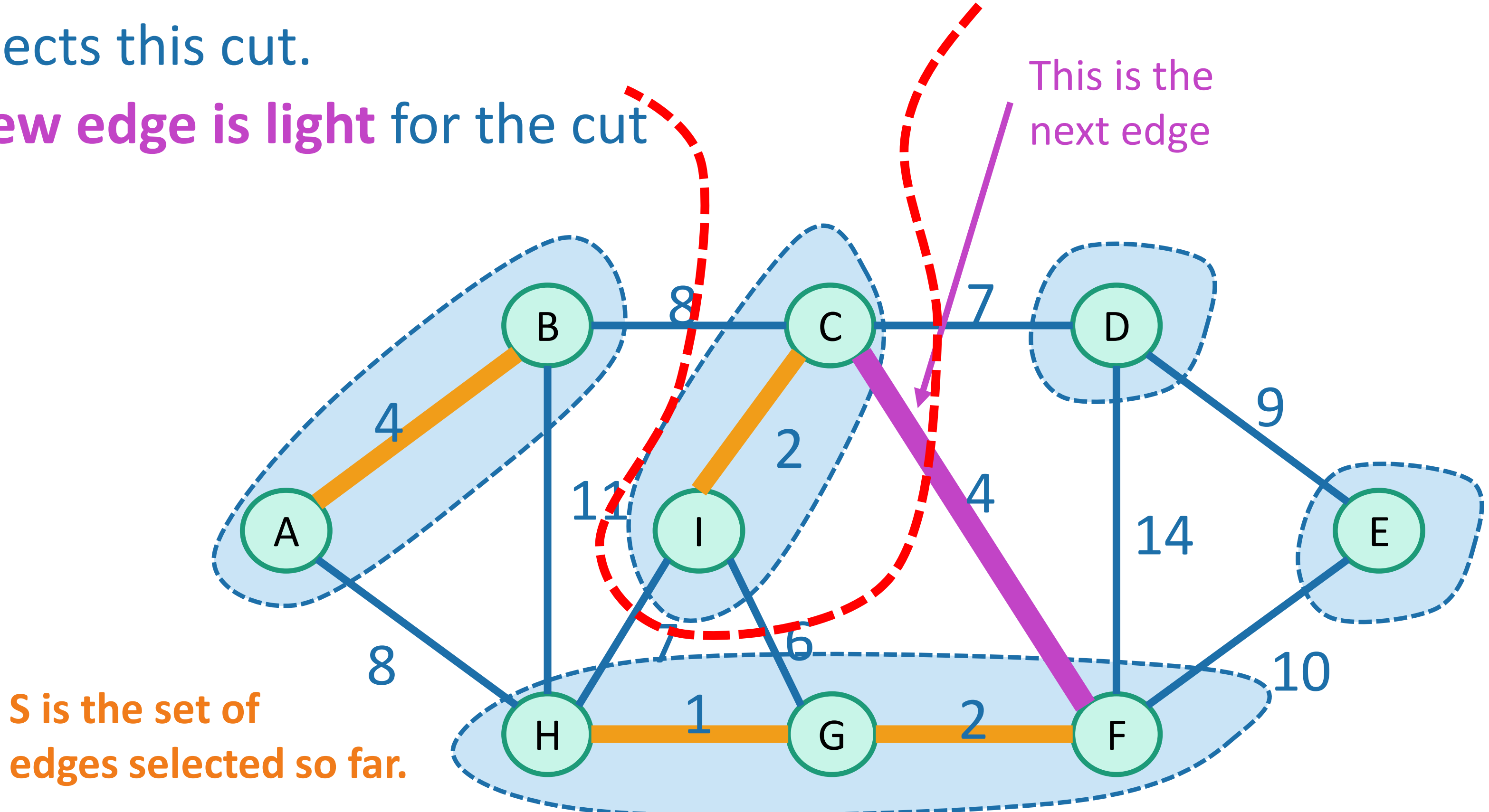
- Assume that our choices **S** so far are **safe.**
  - they don't rule out success
- The **next edge** we add will merge two trees, **T1, T2**



S is the set of
edges selected so far.

Partway through Kruskal

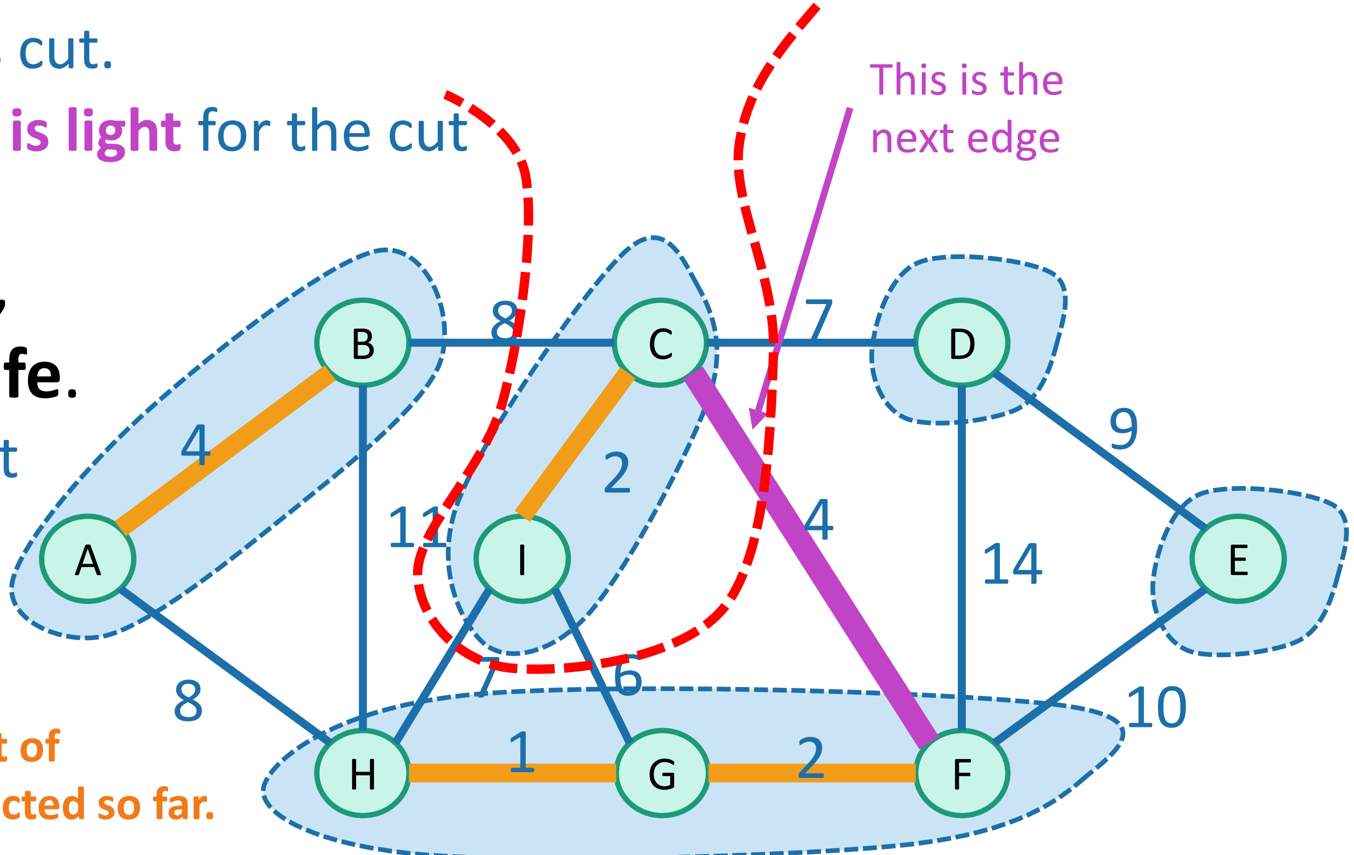
- Assume that our choices **S** so far are **safe.**
  - they don't rule out success
- The **next edge** we add will merge two trees, **T1, T2**
- Consider the cut **{T1, V – T1}.**
  - A respects this cut.
  - Our **new edge is light** for the cut



Partway through Kruskal

- Assume that our choices **S** so far are **safe.**
  - they don't rule out success
- The **next edge** we add will merge two trees, **T1, T2**
- Consider the cut **{T1, V – T1}.**
  - A respects this cut.
  - Our **new edge is light** for the cut

- By the Lemma,
  that edge is safe.
  - it also doesn't rule out success.



S is the set of edges selected so far.

Hooray!

- Our greedy choices **don't rule out success**.
- This is enough (along with an argument by induction) to guarantee correctness of Kruskal's algorithm.

Formally(ish)

This is exactly the same slide that we had for Prim's algorithm.

- Inductive hypothesis:
  - After adding the t'th edge, there exists an MST with the edges added so far.
- Base case:
  - After adding the 0'th edge, there exists an MST with the edges added so far. **YEP.**
- Inductive step:
  - If the inductive hypothesis holds for t (aka, the choices so far are safe), then it holds for t+1 (aka, the next edge we add is safe).
  - **That's what we just showed.**
- Conclusion:
  - After adding the n-1'st edge, there exists an MST with the edges added so far.
  - At this point we have a spanning tree, so it better be minimal.

Two questions

1. Does it work?
   - That is, does it actually return a MST?
     - **Yes**
2. How do we actually implement this?
   - the pseudocode above says “slowKruskal”…
     - **Using a union-find data structure!**

What have we learned?

- Kruskal's algorithm greedily grows a forest
- It finds a Minimum Spanning Tree in time O(mlog(n))
  - if we implement it with a Union-Find data structure

- To prove it worked, we followed the same recipe for greedy algorithms we saw last time.
  - Show that, at every step, we **don't rule out success.**

Comparison of Kruskal and Prims

Compare and contrast

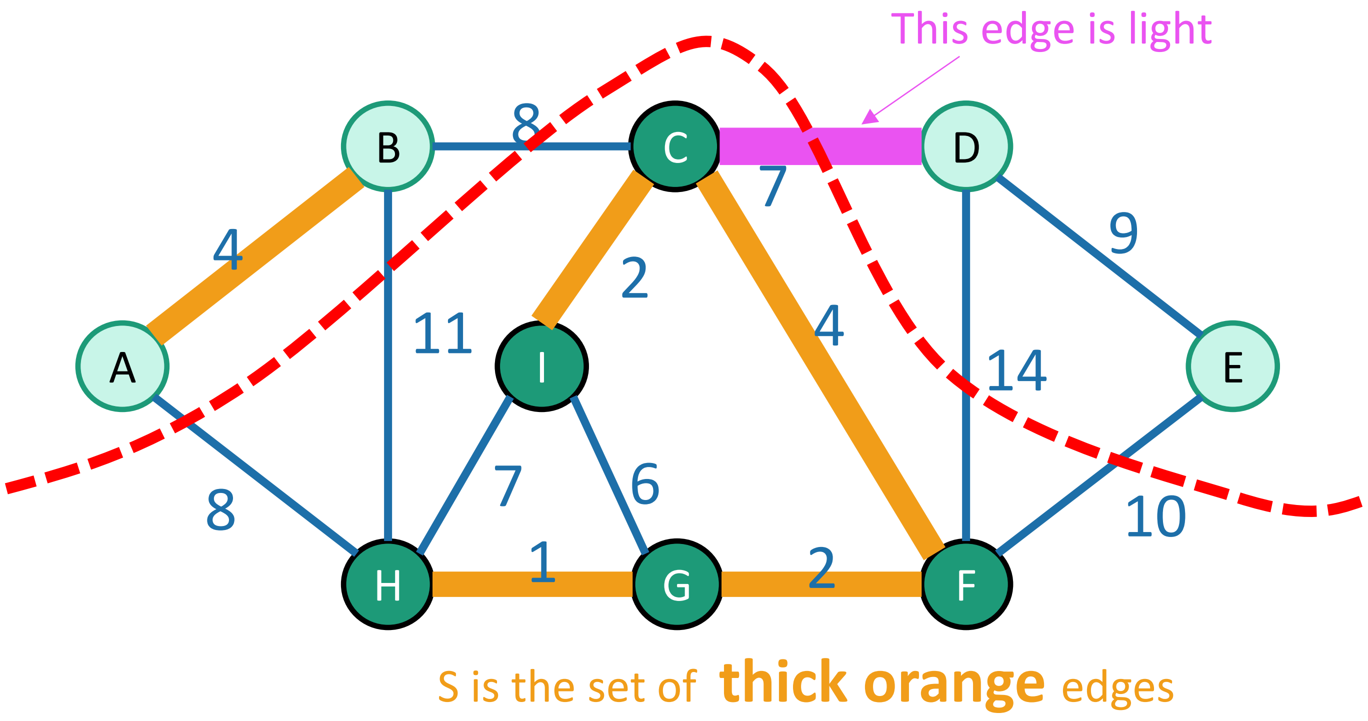
- Prim:
  - Grows a tree.
  - Time O(mlog(n)) with a red-black tree
- Kruskal:
  - Grows a forest.
  - Time O(mlog(n)) with a union-find data structure
  - If you can do radixSort on the edge weights, morally O(m)

Prim might be a better idea on dense graphs

Kruskal might be a better idea on sparse graphs if you can radixSort edge weights

Both Prim and Kruskal

- Greedy algorithms for MST.
- Similar reasoning:
  - Optimal substructure: subgraphs generated by cuts.
  - The way to make safe choices is to choose light edges crossing the cut.



Can we do better?

State-of-the-art MST on connected undirected graphs

- Karger-Klein-Tarjan 1995:
  - O(m) time randomized algorithm
- Chazelle 2000:
  - O(m$\cdot\, \alpha(n)$) time deterministic algorithm
- Pettie-Ramachandran 2002:
  - O(The optimal number of comparisons N*(n,m) you need to solve the problem, whatever that is…) time deterministic algorithm

What is this number?
Do we need that silly $\alpha(n)$?
Open questions!

Recap

- Two algorithms for Minimum Spanning Tree
  - Prim's algorithm
  - Kruskal's algorithm
- Both are (more) examples of greedy algorithms!
  - Make a **series of choices**.
  - Show that at each step, your choice **does not rule out success.**
  - At the end of the day, you haven't ruled out success, so **you must be successful**.