Strategies for Algorithm Design

Greedy Algorithms

Minimum Spanning Tree (MST)

Last time

- Greedy algorithms
  - Make a series of choices.
    - Choose this activity, then that one, ..
    - Never backtrack.
  - Show that, at each step, your choice does not rule out success.
    - At every step, there exists an optimal solution consistent with the choices we've made so far.
  - At the end of the day:
    - you've built only one solution,
    - never having ruled out success,
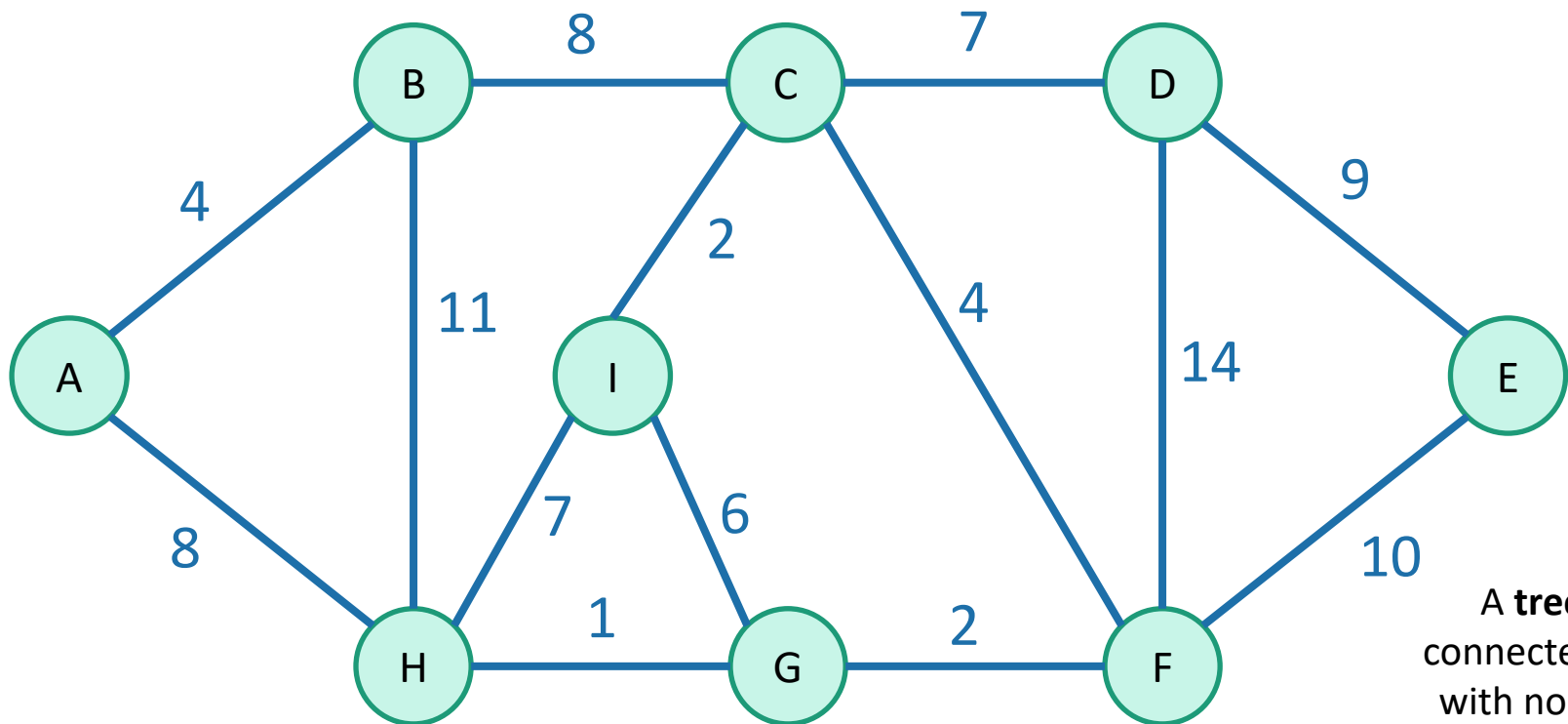   - **so your solution must be correct.**

Today

- Greedy algorithms for Minimum Spanning Tree.

- Agenda:
  1. What is a Minimum Spanning Tree?
  2. Short break to introduce some graph theory tools
  3. Prim's algorithm
  4. Kruskal's algorithm

Minimum Spanning Tree
Say we have an undirected weighted graph

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

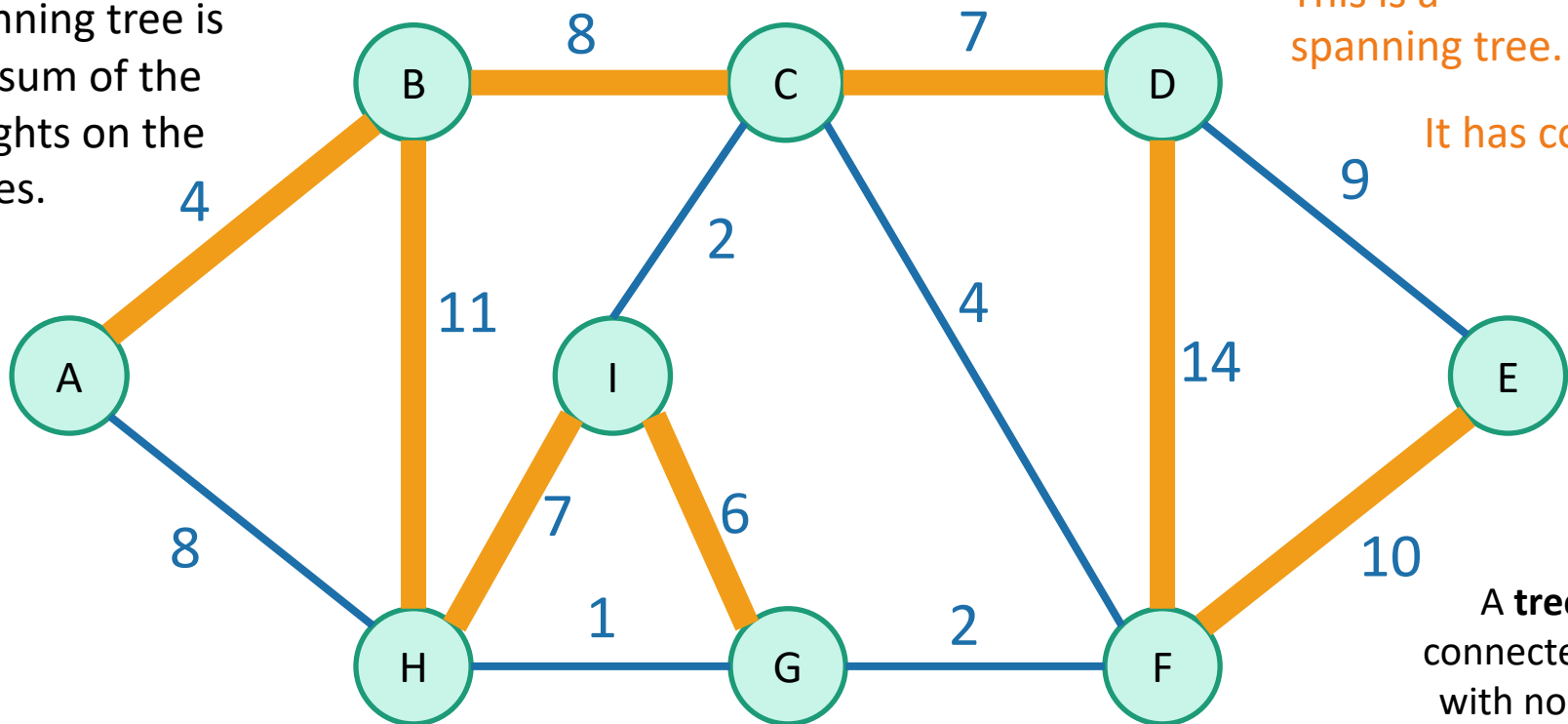A **spanning tree** is a **tree** that connects all of the vertices.

4

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.
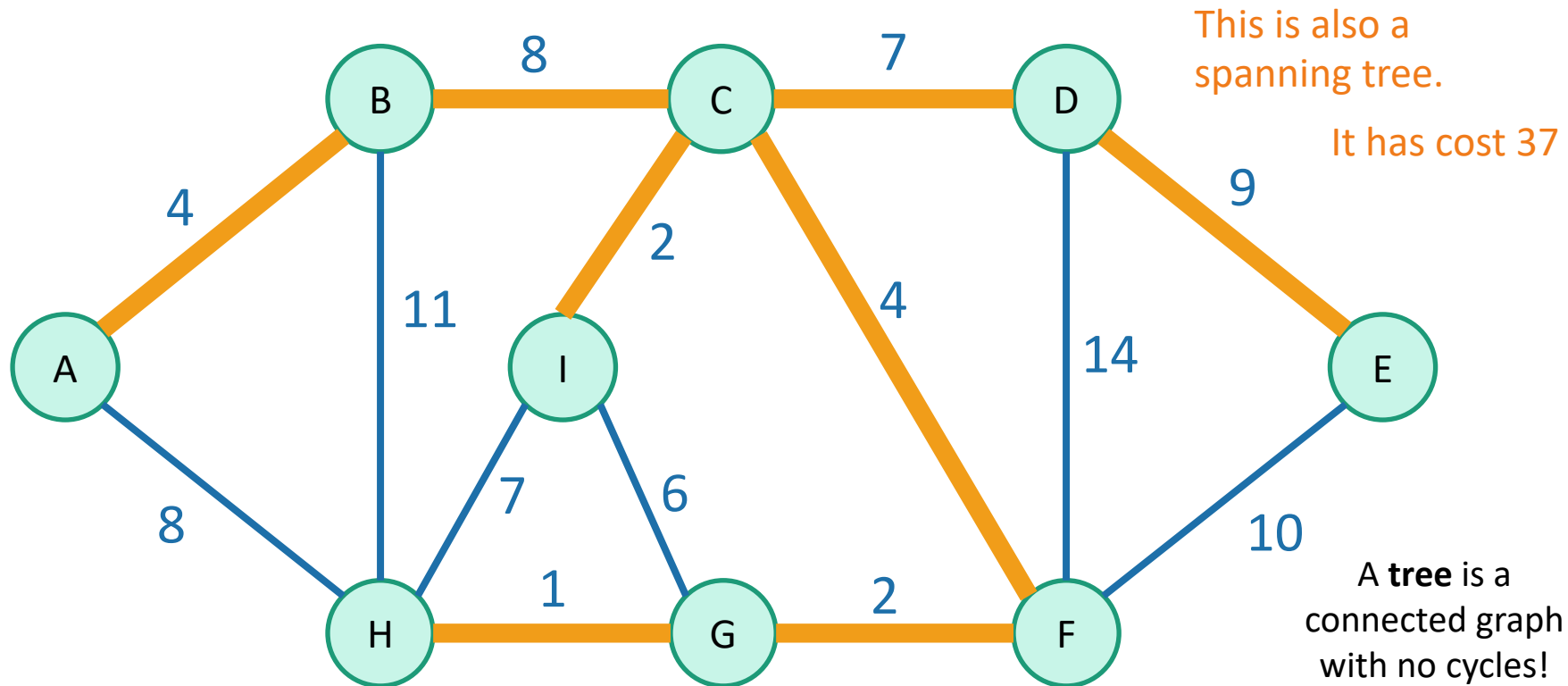
This is a spanning tree.

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.
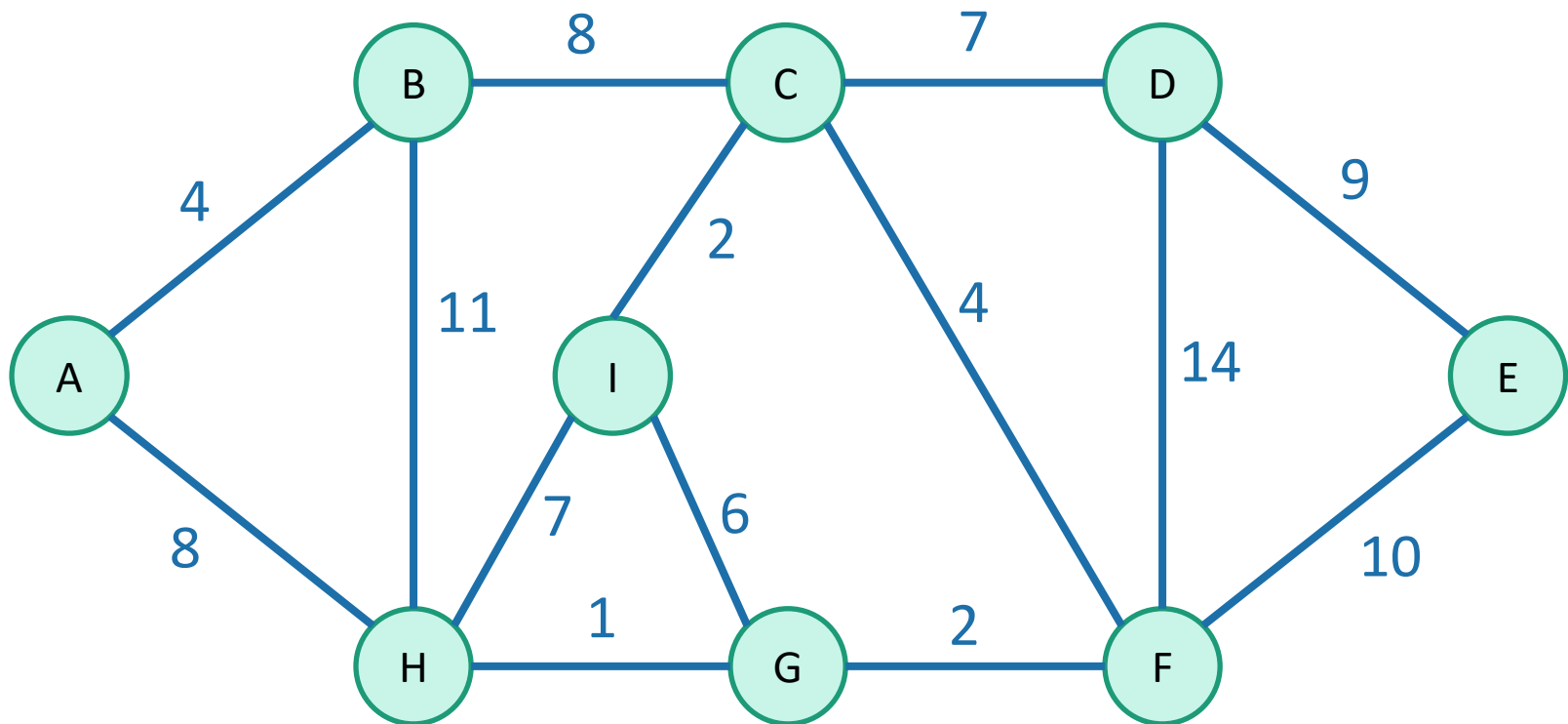
Minimum Spanning Tree
Say we have an undirected weighted graph

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

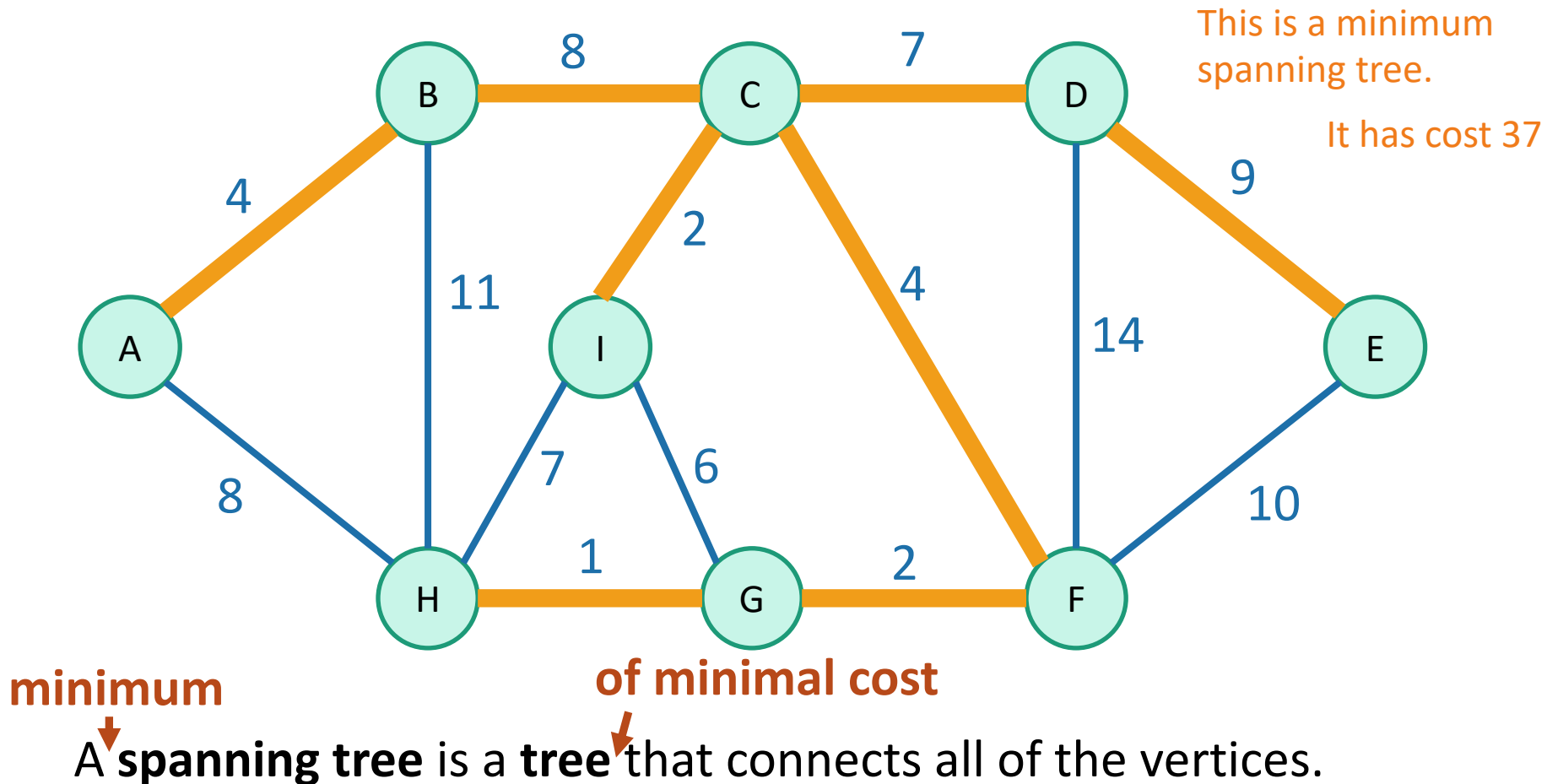Say we have an undirected weighted graph
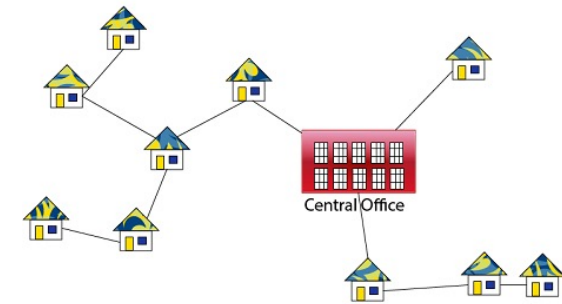

minimum

of minimum 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

Wiring : Better Approach

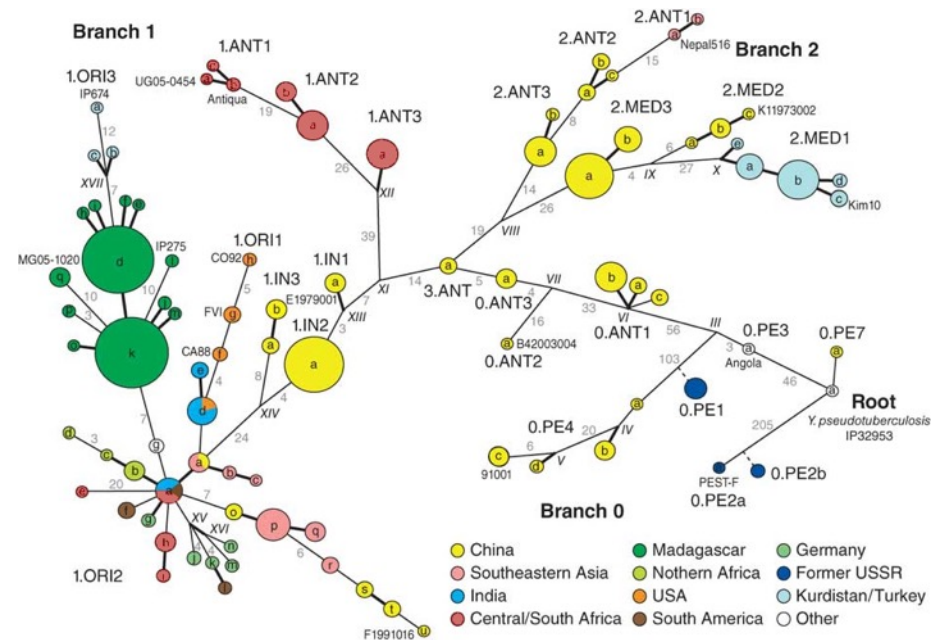Minimize the total length of wire connecting the customers

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

9

How to find an MST?

- Today we'll see two greedy algorithms.
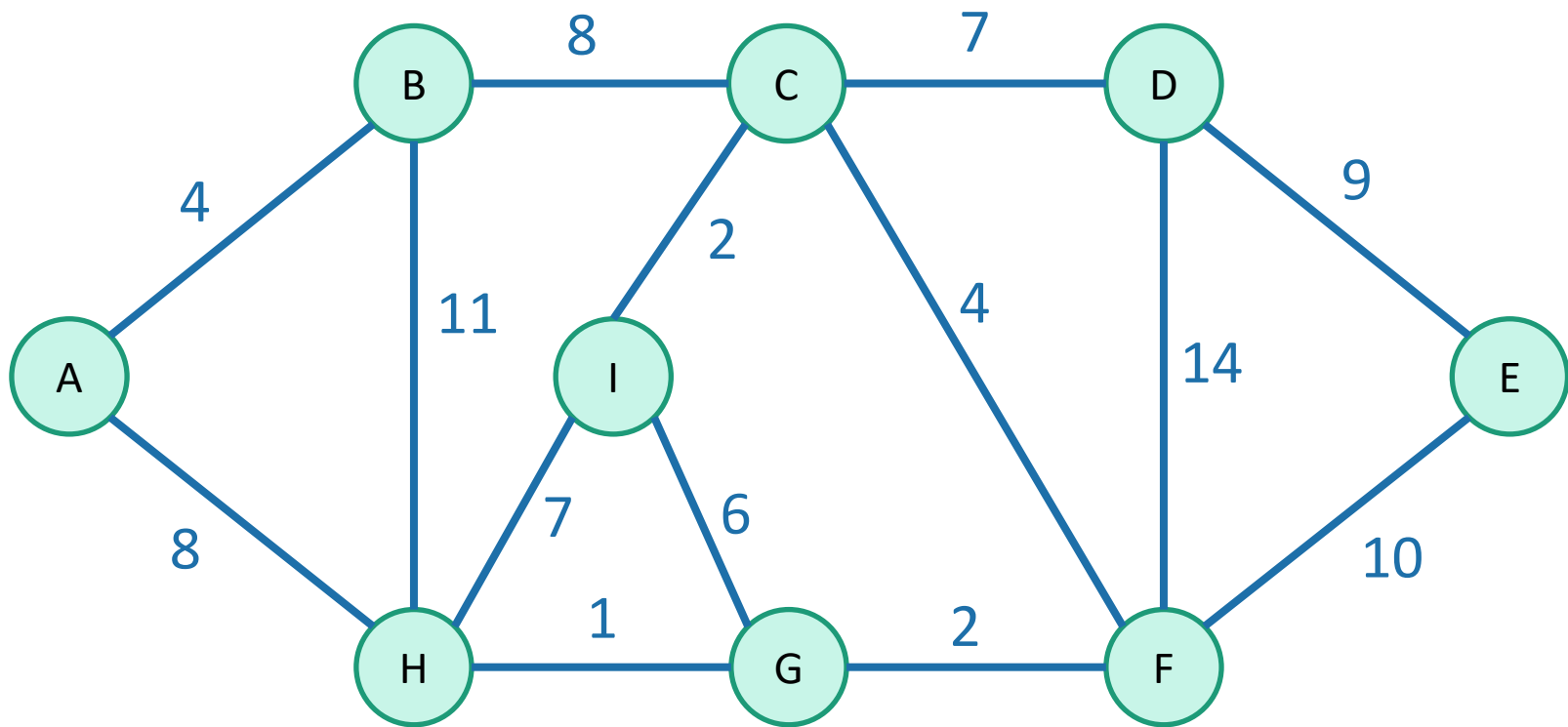- In order to prove that these greedy algorithms work, we'll show something like:

Suppose that our choices so far

are consistent with an MST.

Then the next greedy choice that we make

is still consistent with an MST.

- This is not the only way to prove that these algorithms work!

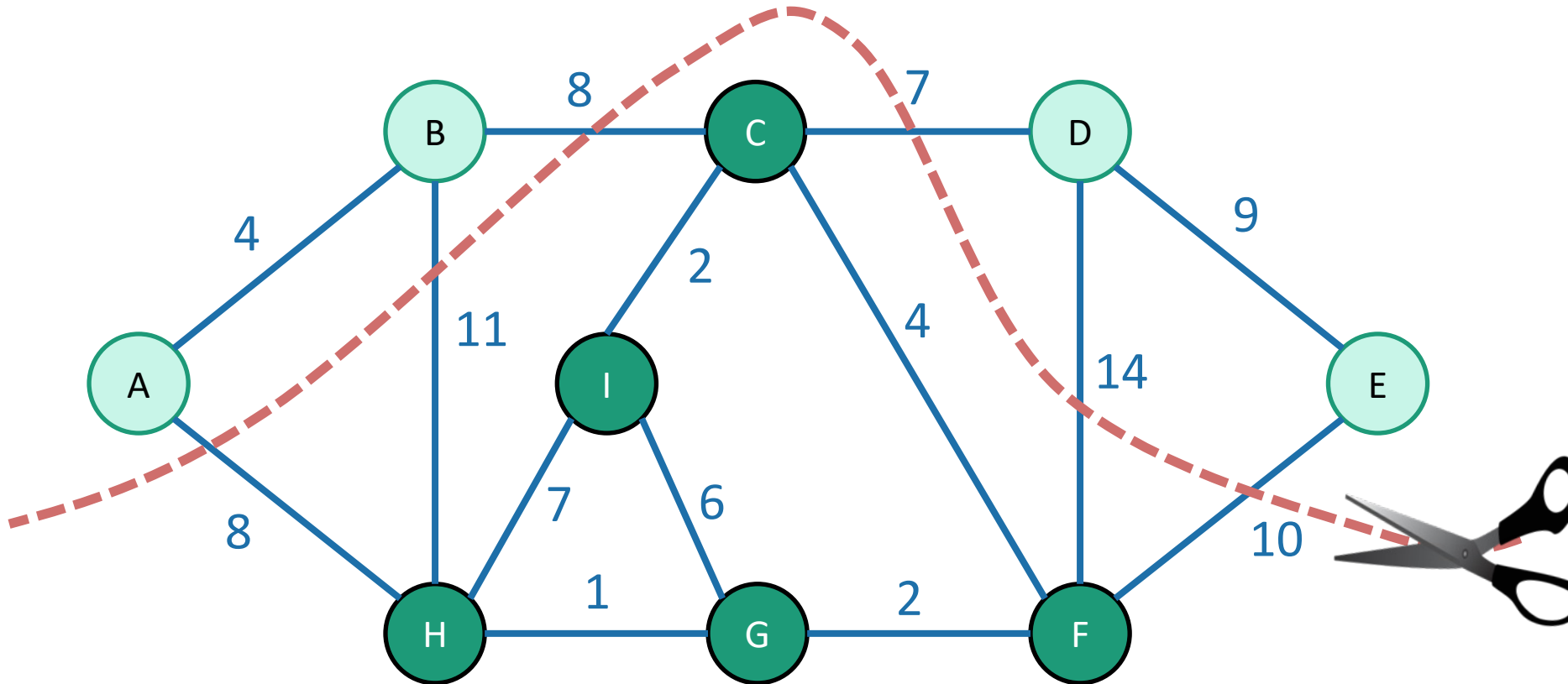# Let's brainstorm some greedy algorithms!
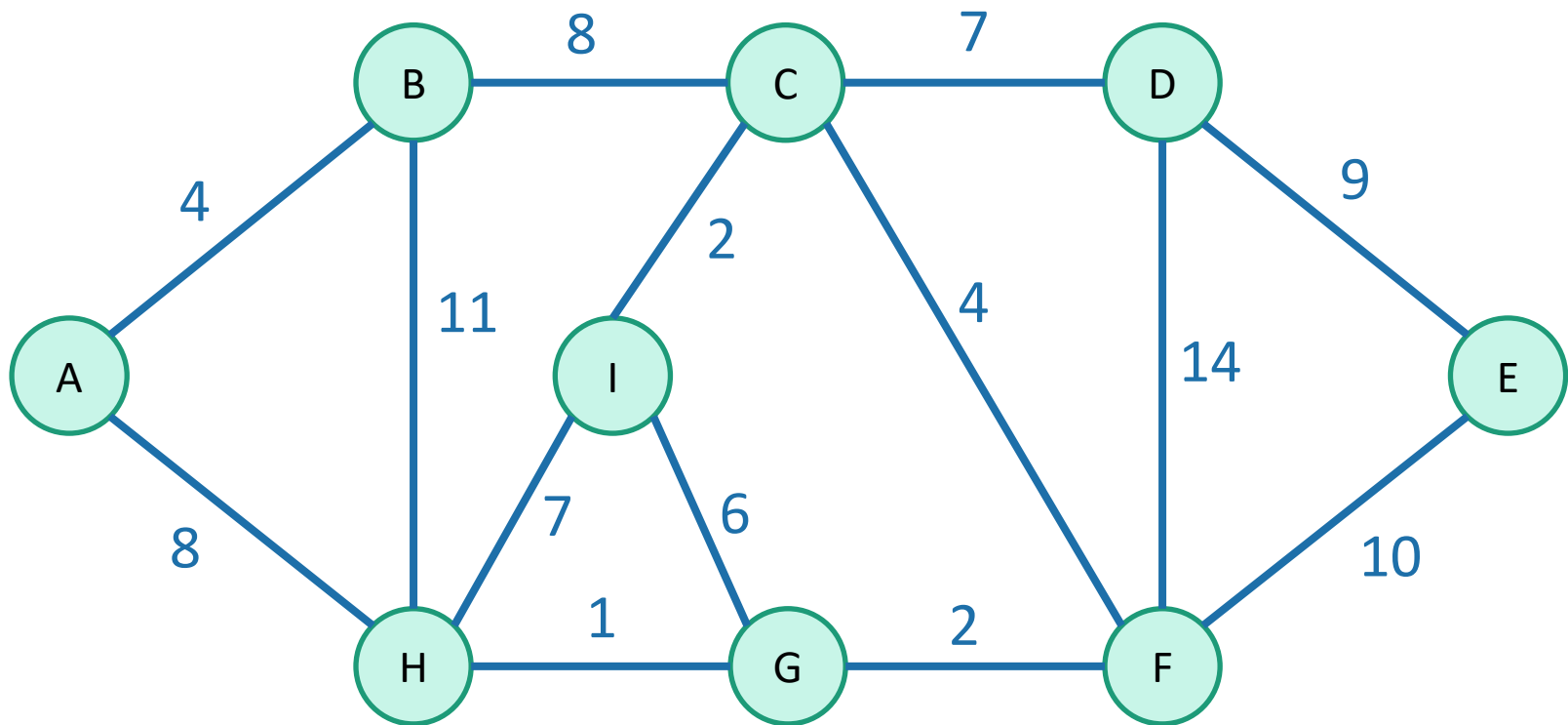
GENERIC-MST(G, w)

1   $A = \emptyset$
2   **while** A does not form a spanning tree
3   find an edge (u, v) that is safe for A
4         $A = A \cup \{(u, v)\}$
5   **return** A

Brief aside

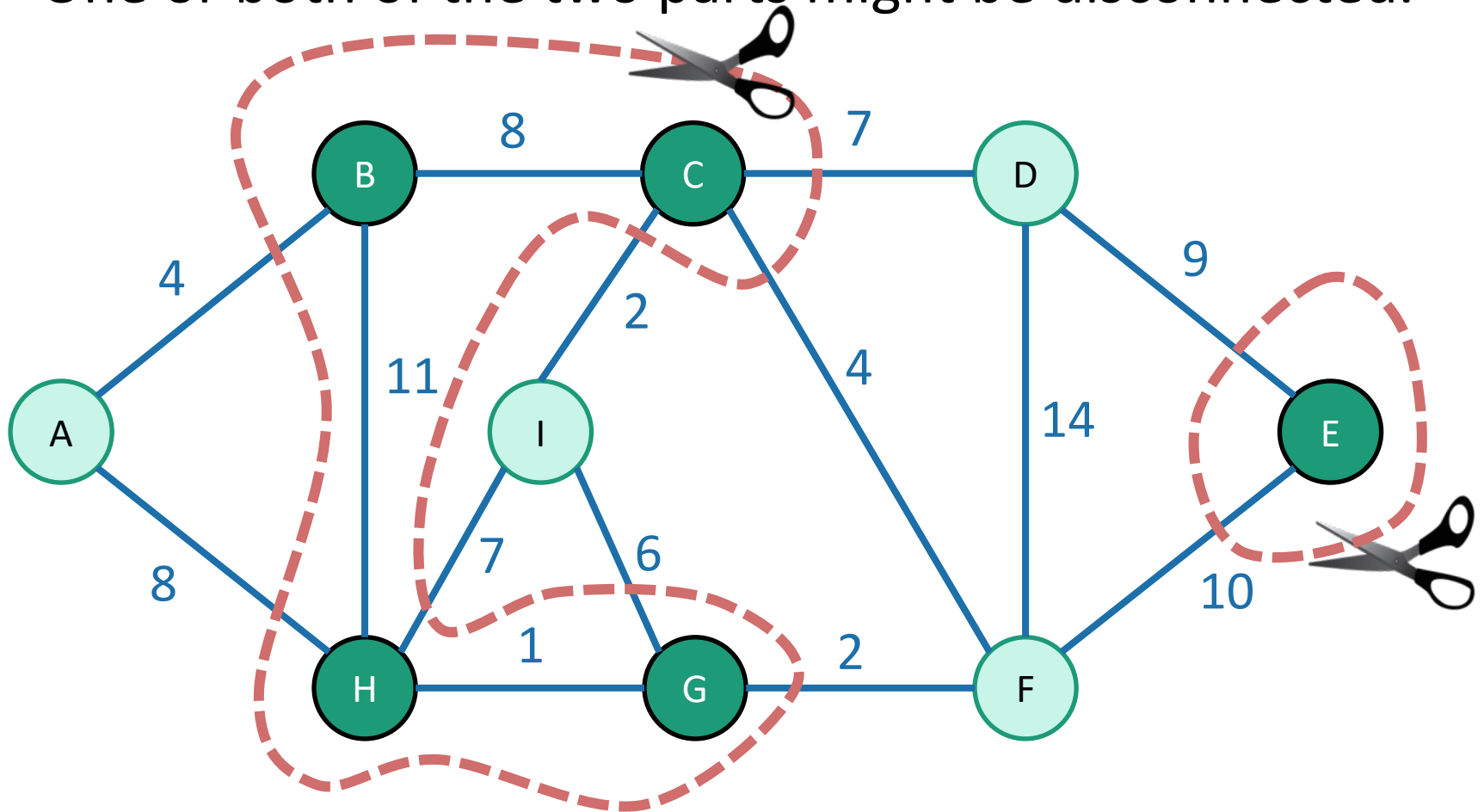for a discussion of cuts in graphs!

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}"**
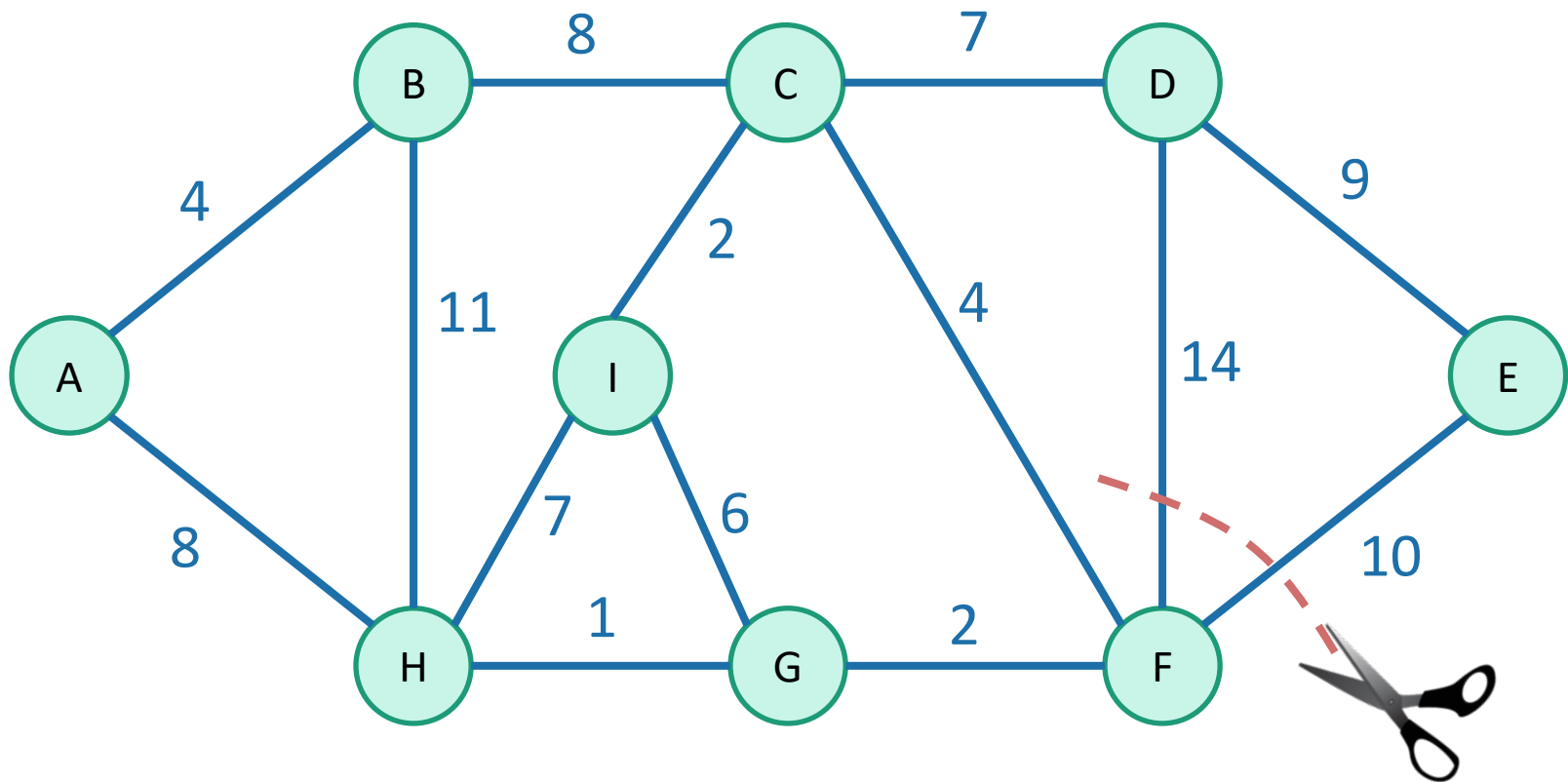
Cuts in graphs

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

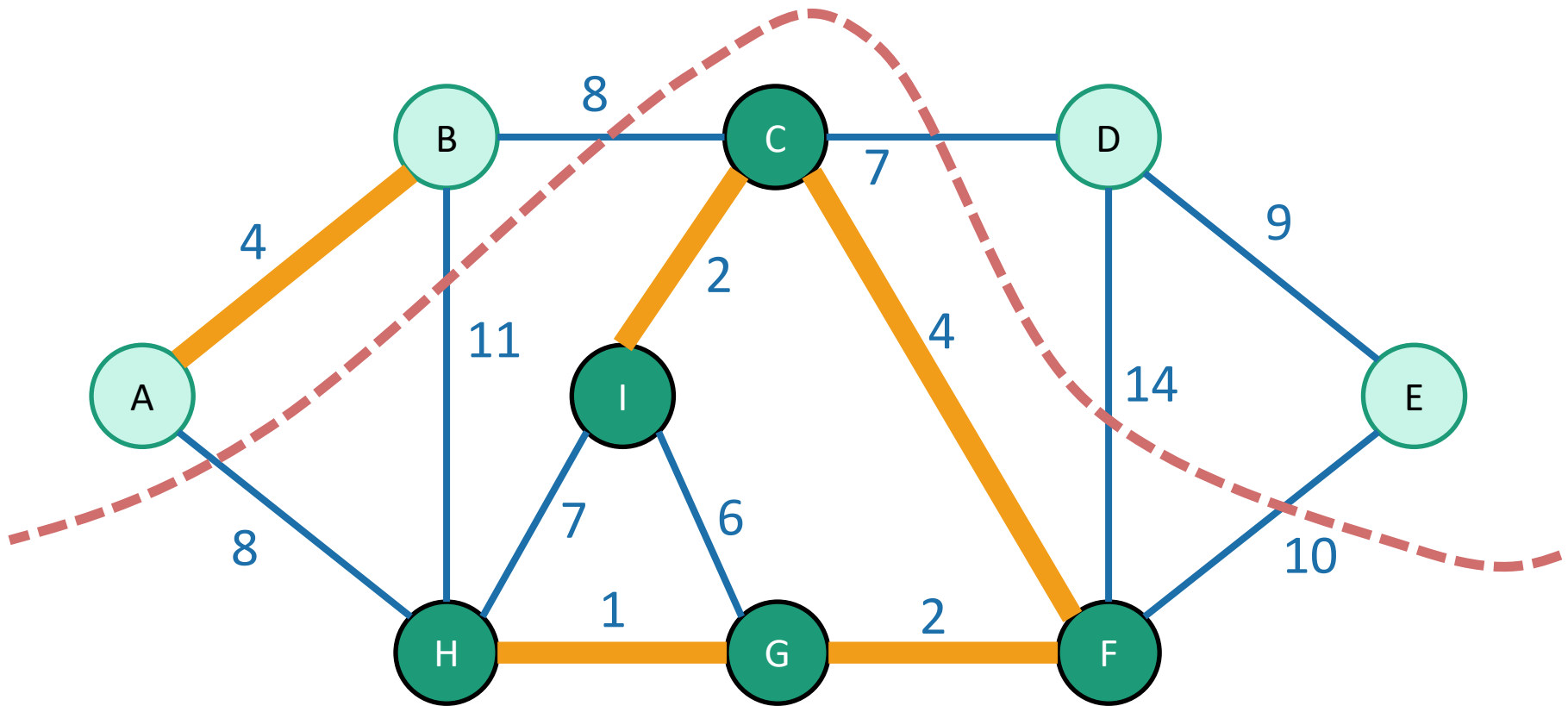This is the cut **"{B,C,E,G,H} and {A,D,I,F}"**

Cuts in graphs

- This is ***not*** a cut. Cuts are partitions of vertices.
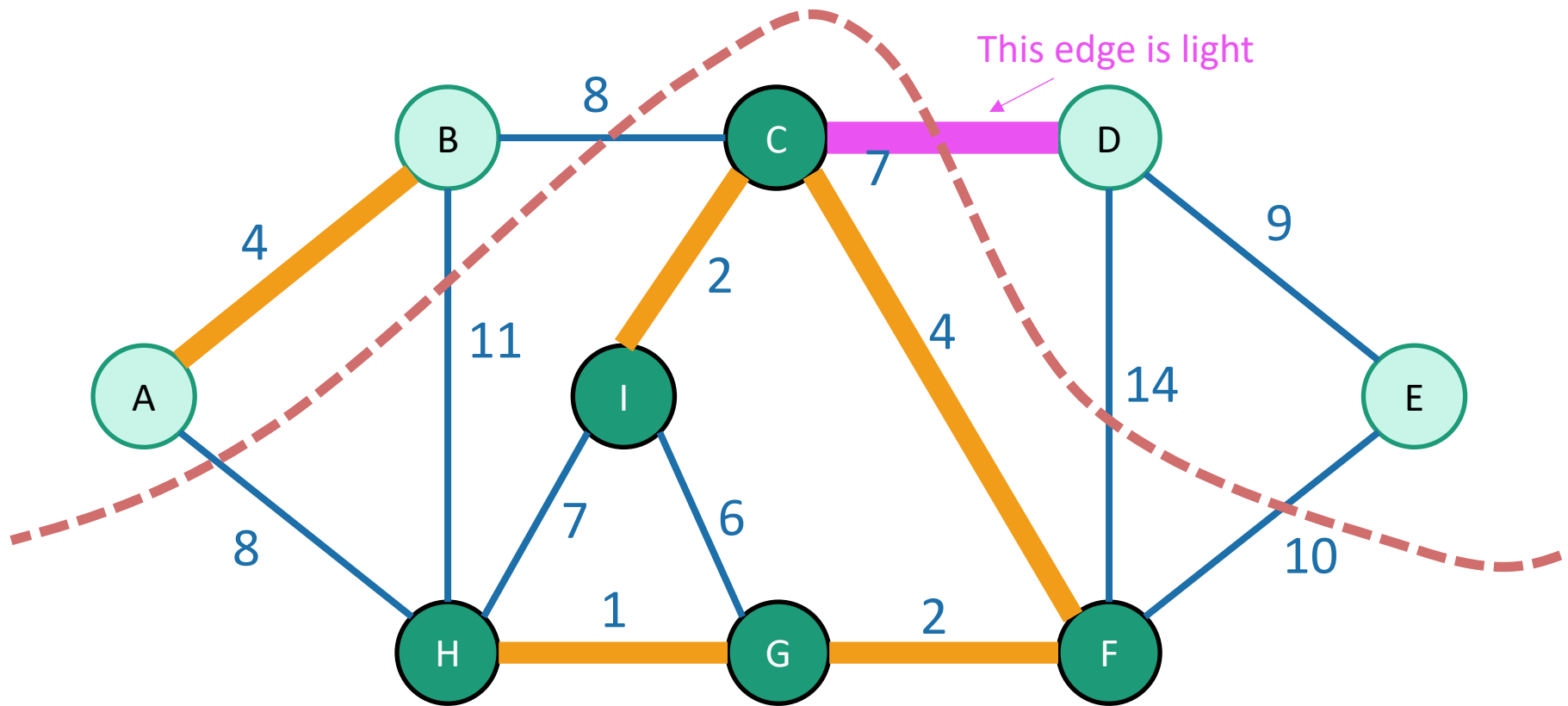
Let S be a set of edges in G

- 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
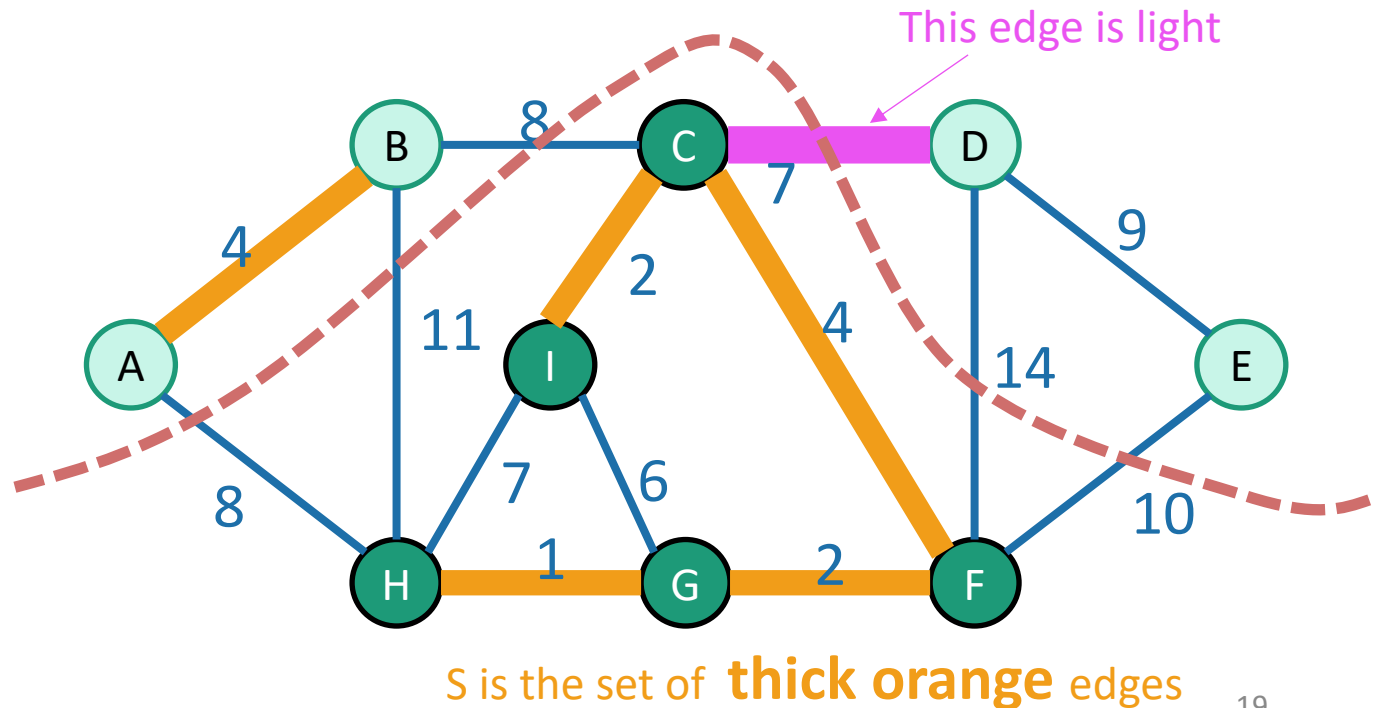
Let S be a set of edges in G

- 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.

This edge is light

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}}

This edge is light

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

19

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.
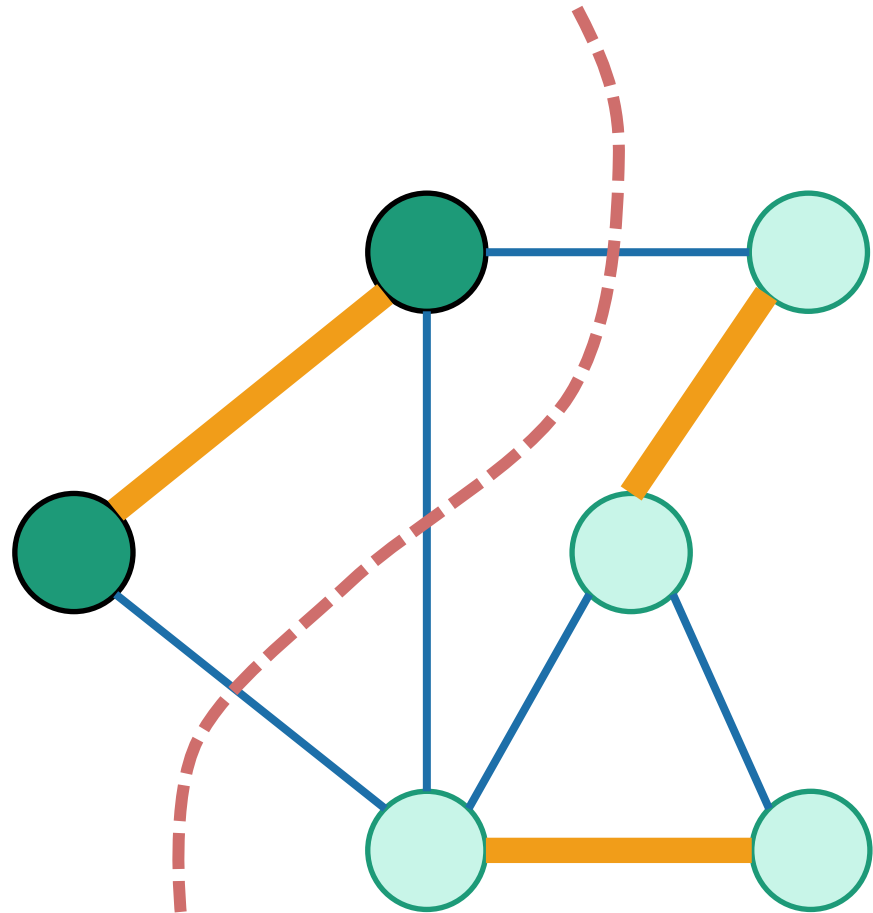- Then there is an MST containing S ∪ {{u,v}}

Aka:

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

It's "safe" to add this edge!
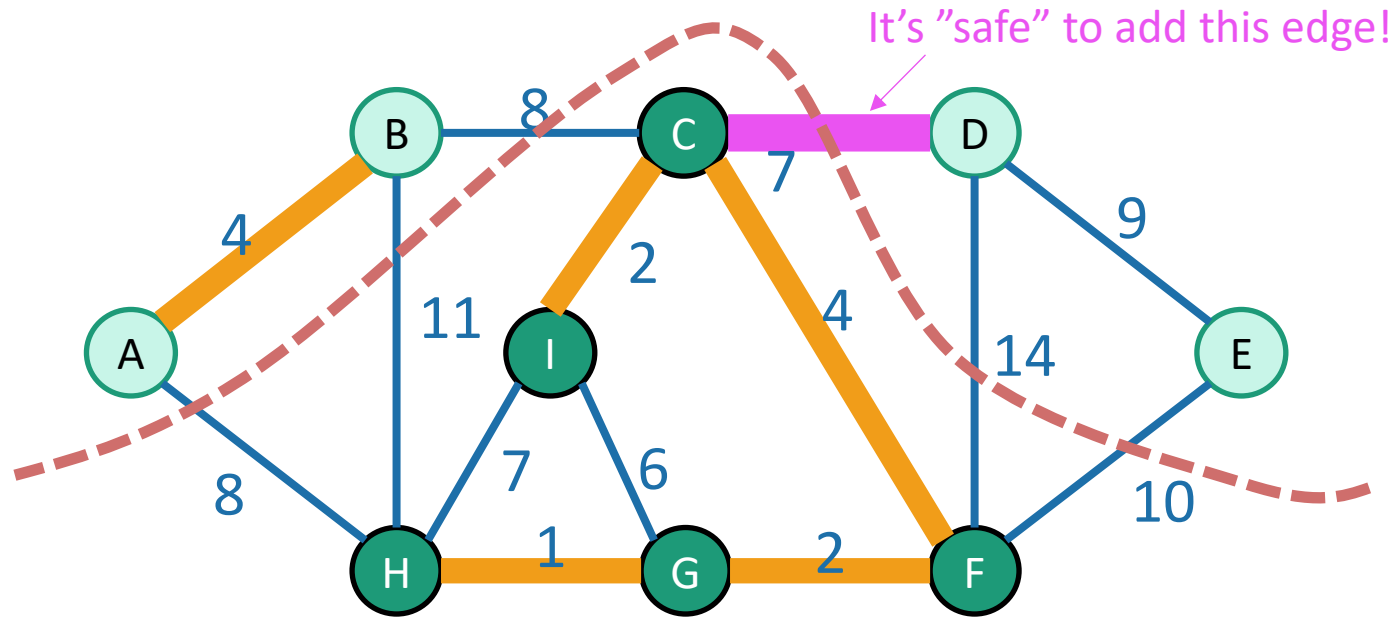
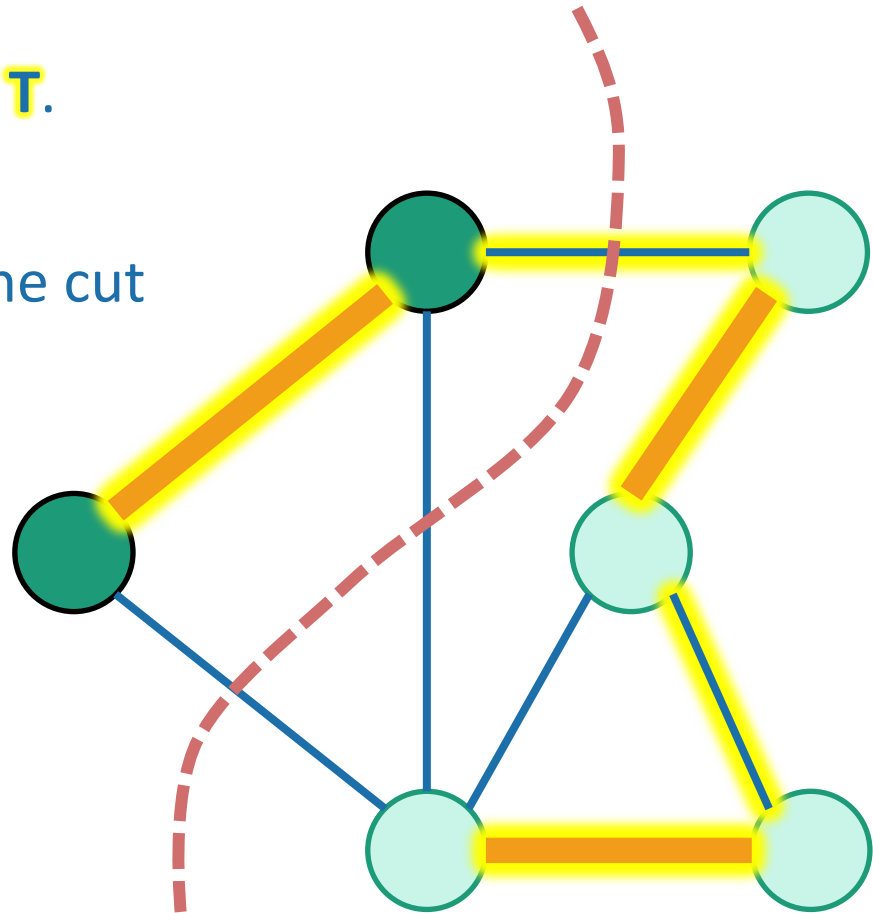S is the set of **thick orange** edges

Proof of Lemma

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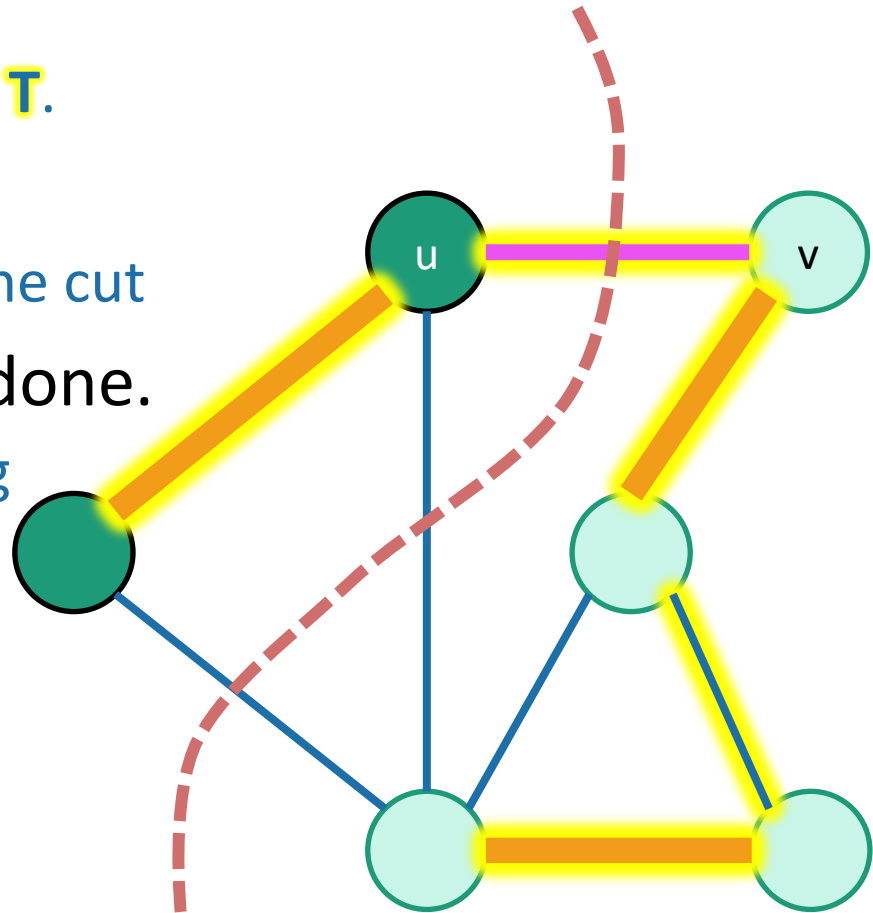 - a **cut** that respects **S**

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

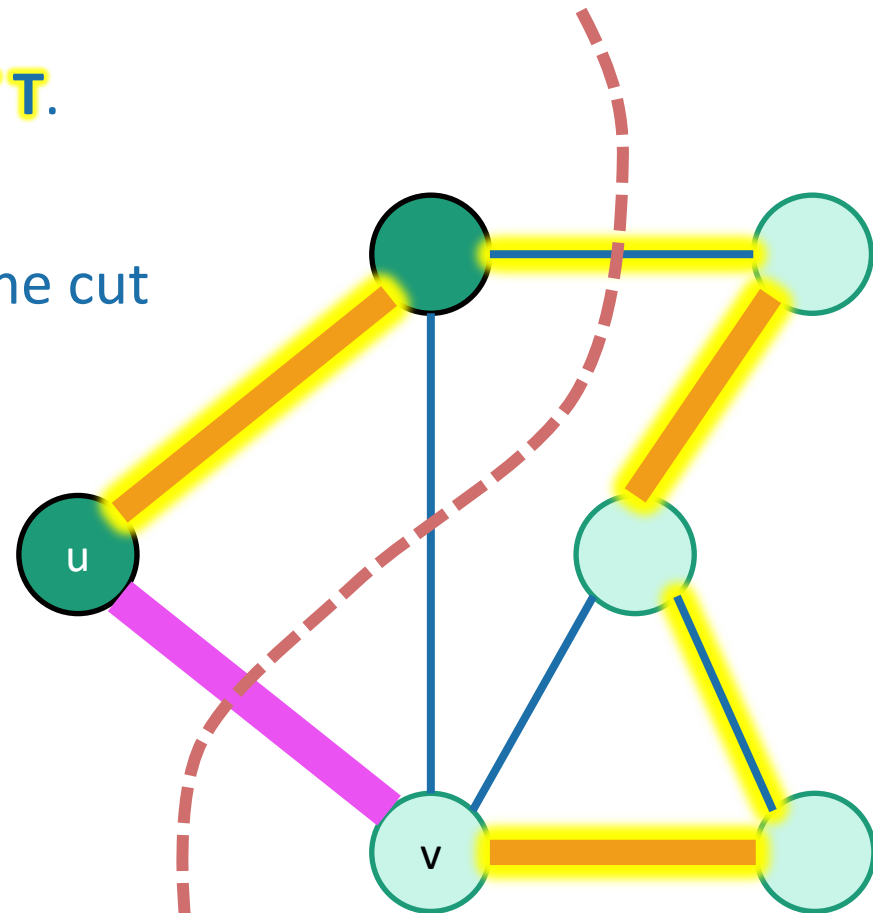# Proof of Lemma

- 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
- If **{u,v}** is in T, we are done.
  - T is an MST containing both {u,v} and S.
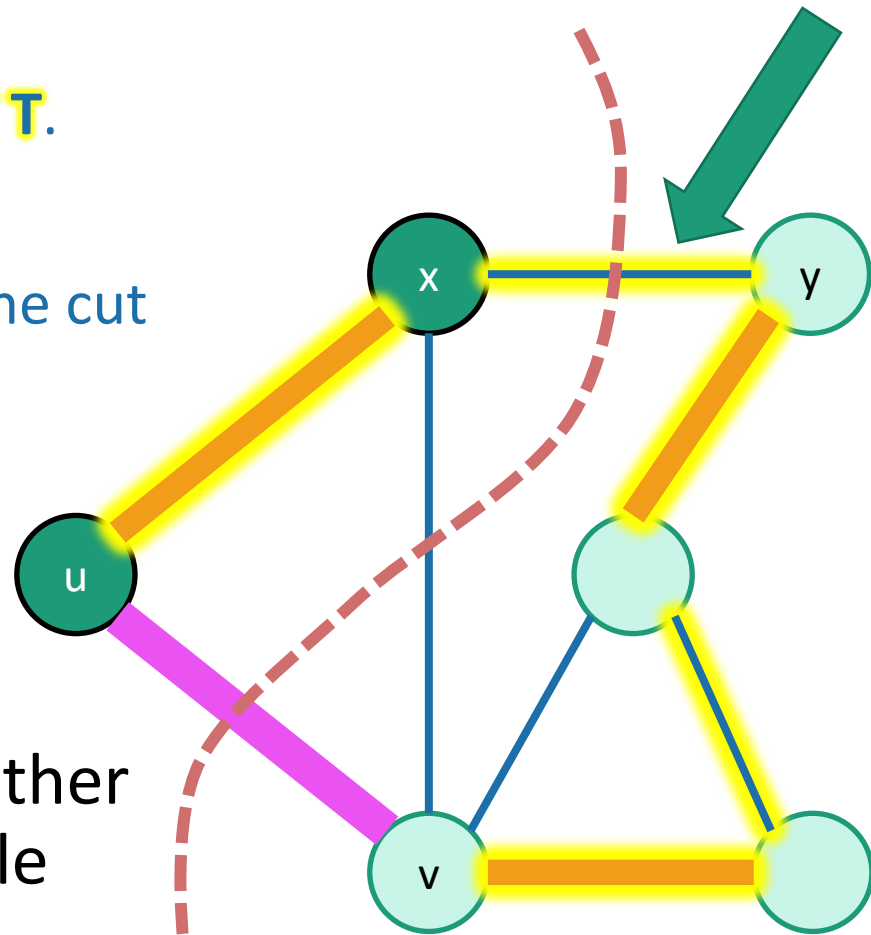

23

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
- Say **{u,v}** is not in **T**.
- Note that adding **{u,v}** to **T** will make a cycle.

24

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

- Say **{u,v}** is not in **T**.

- Note that adding **{u,v}** to **T** will make a cycle.

- There is at least one other edge, **{x,y}**, in this cycle crossing the cut.

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

25

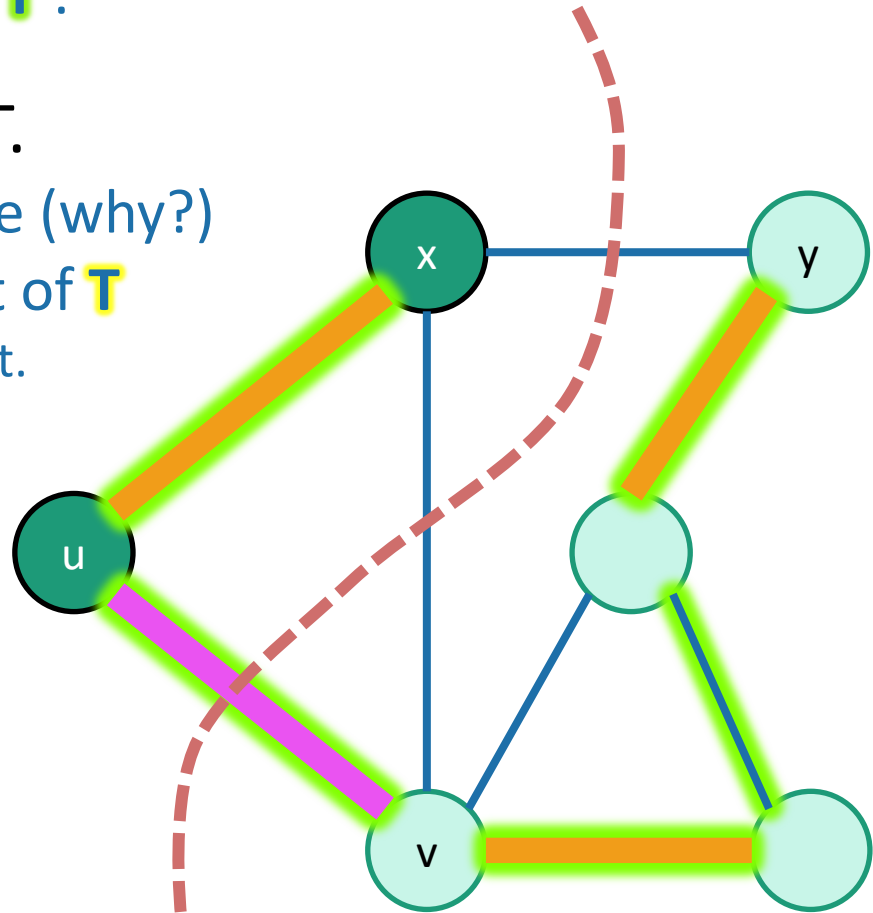# Proof of Lemma ctd.

- Consider swapping {u,v} for {x,y} in **T**.
  - Call the resulting tree **T'**.
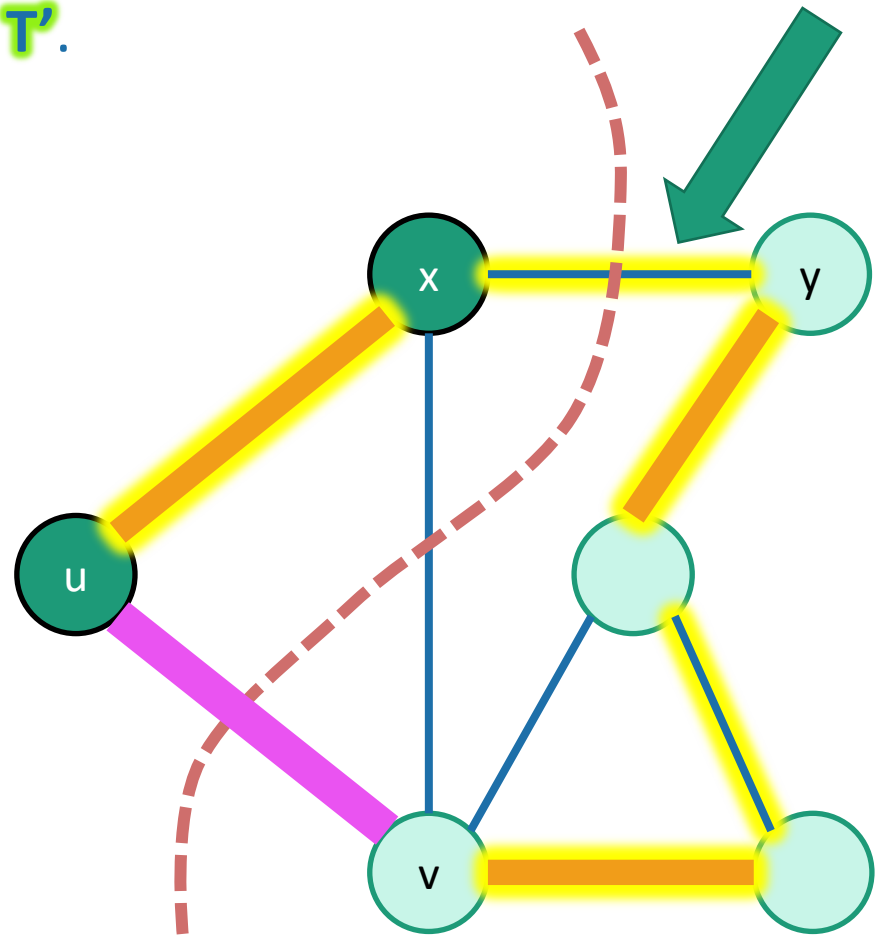
Proof of Lemma ctd.

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

- **Claim**: **T'** is still an MST.
  - It is still a spanning tree (why?)
  - 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 S and {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}}

This edge is light

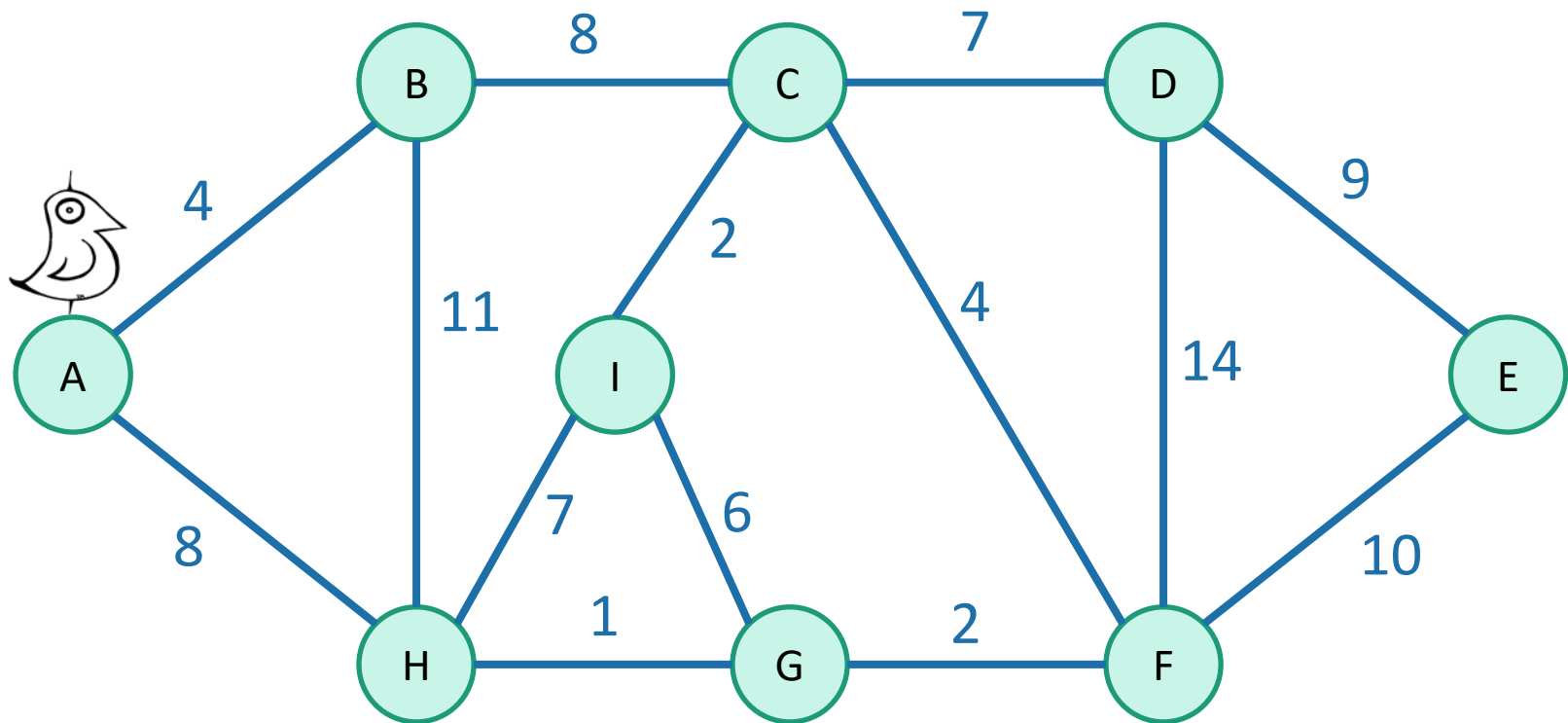S is the set of **thick orange** edges
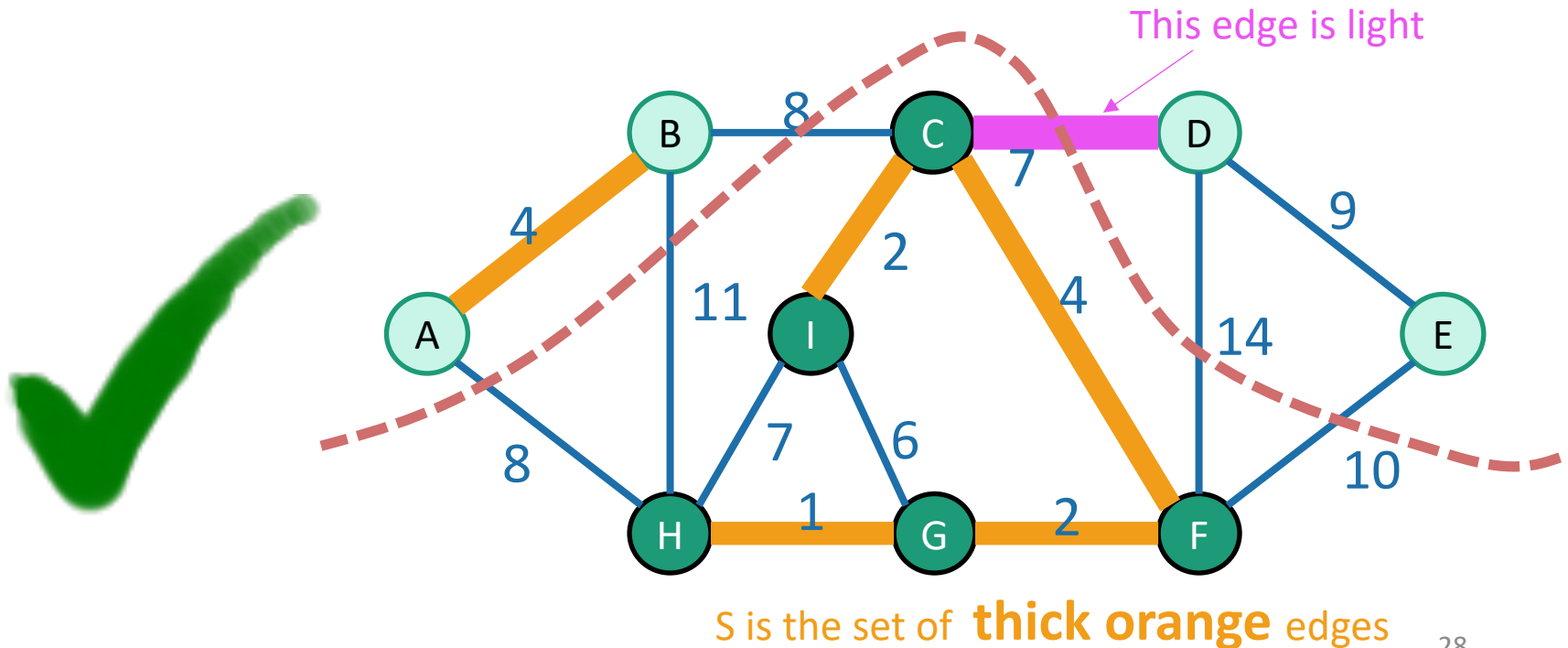
End aside

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**.
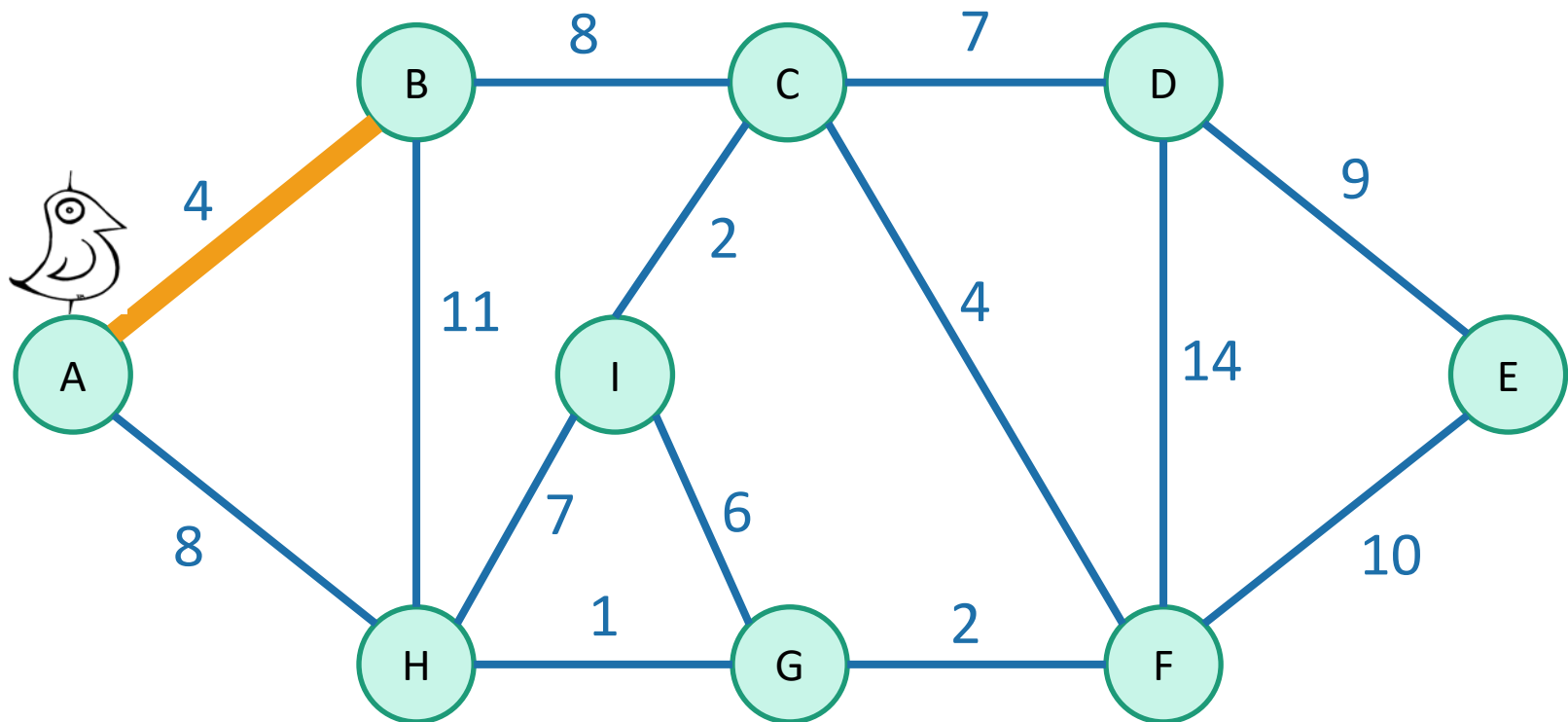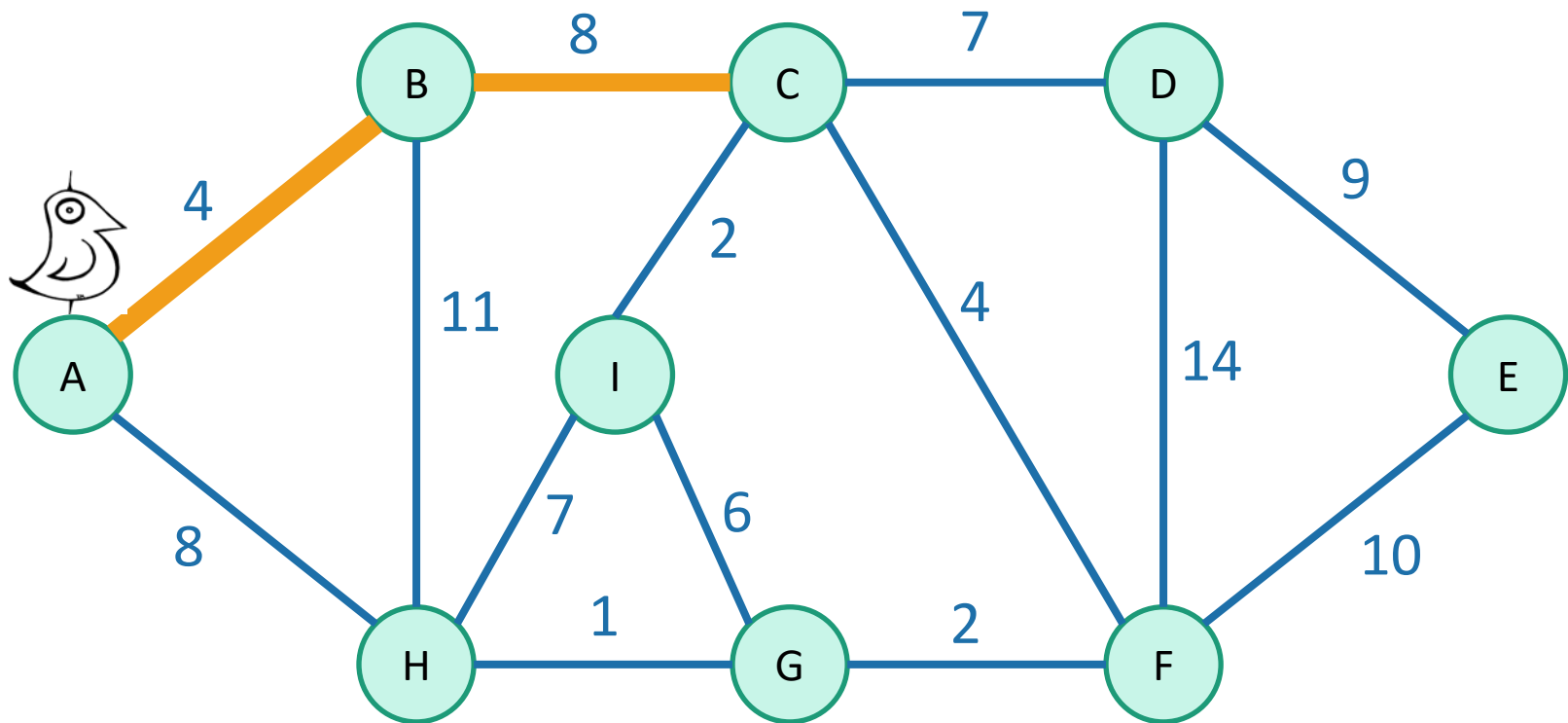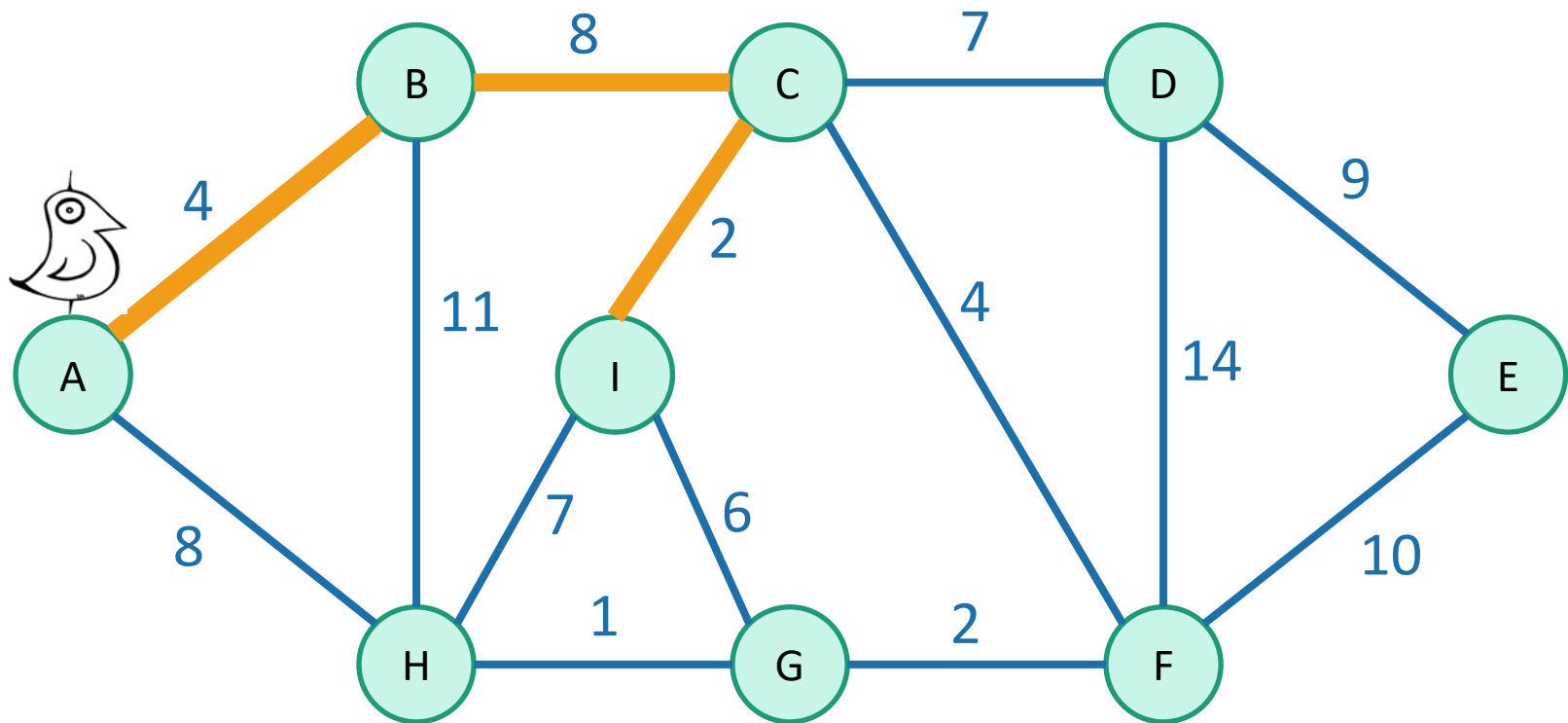 - **Keep going** until we have an MST.

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.
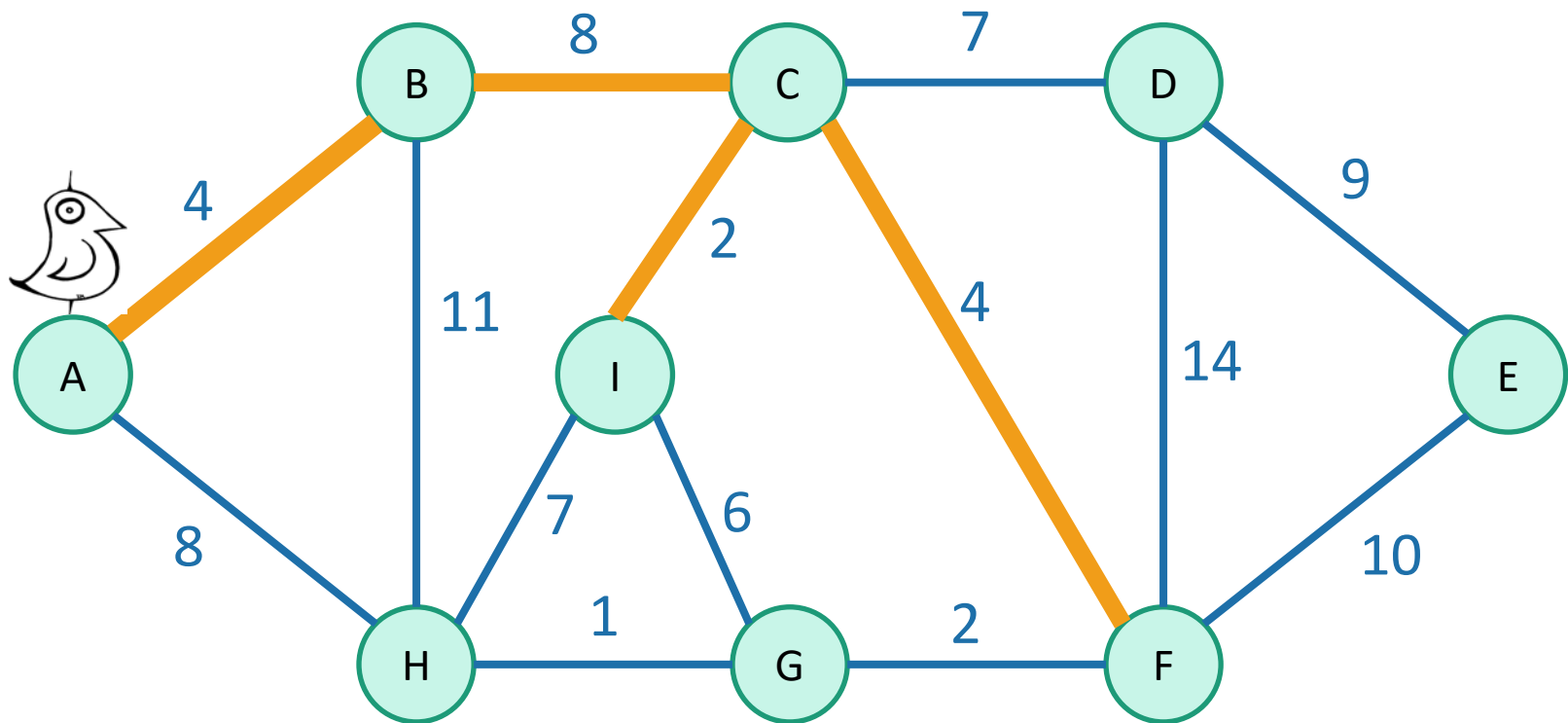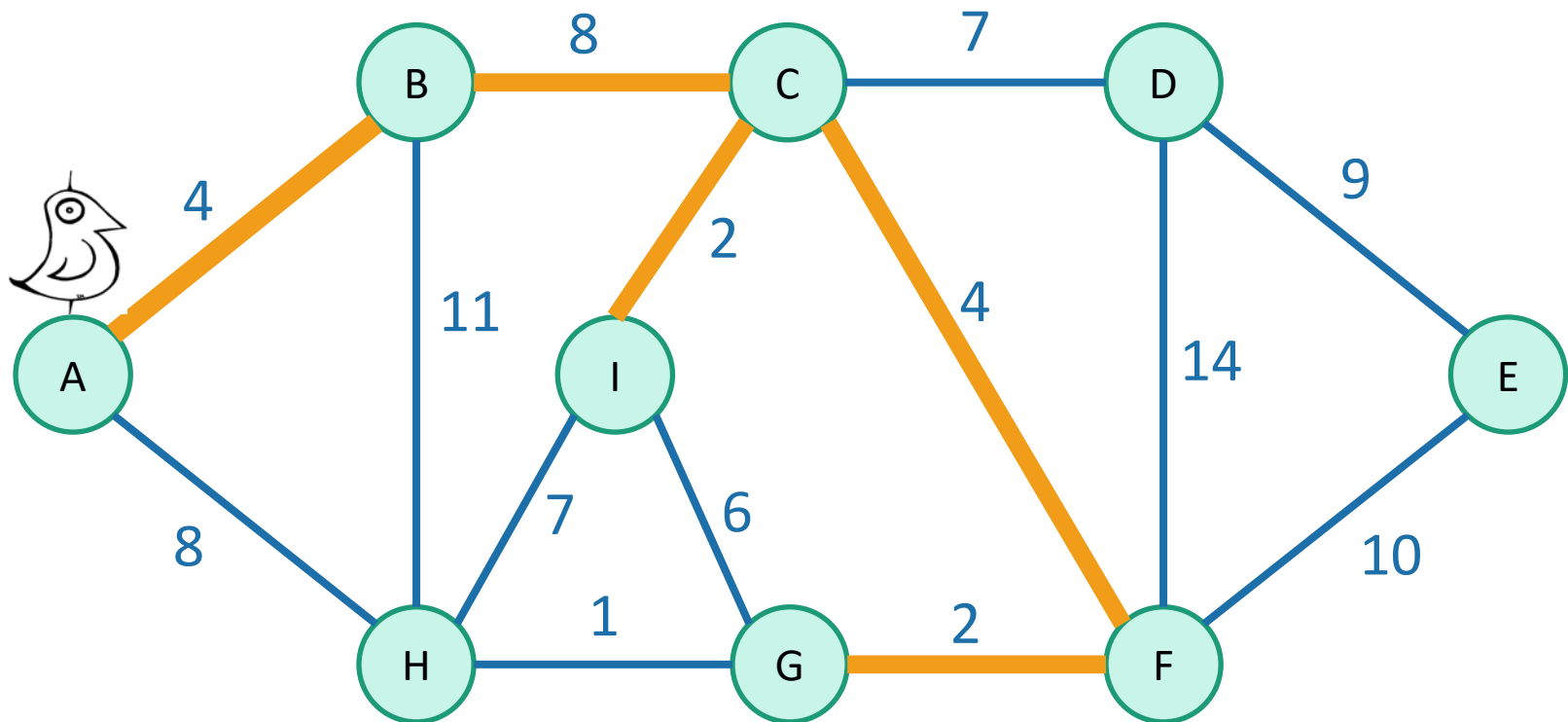
Idea 1

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.
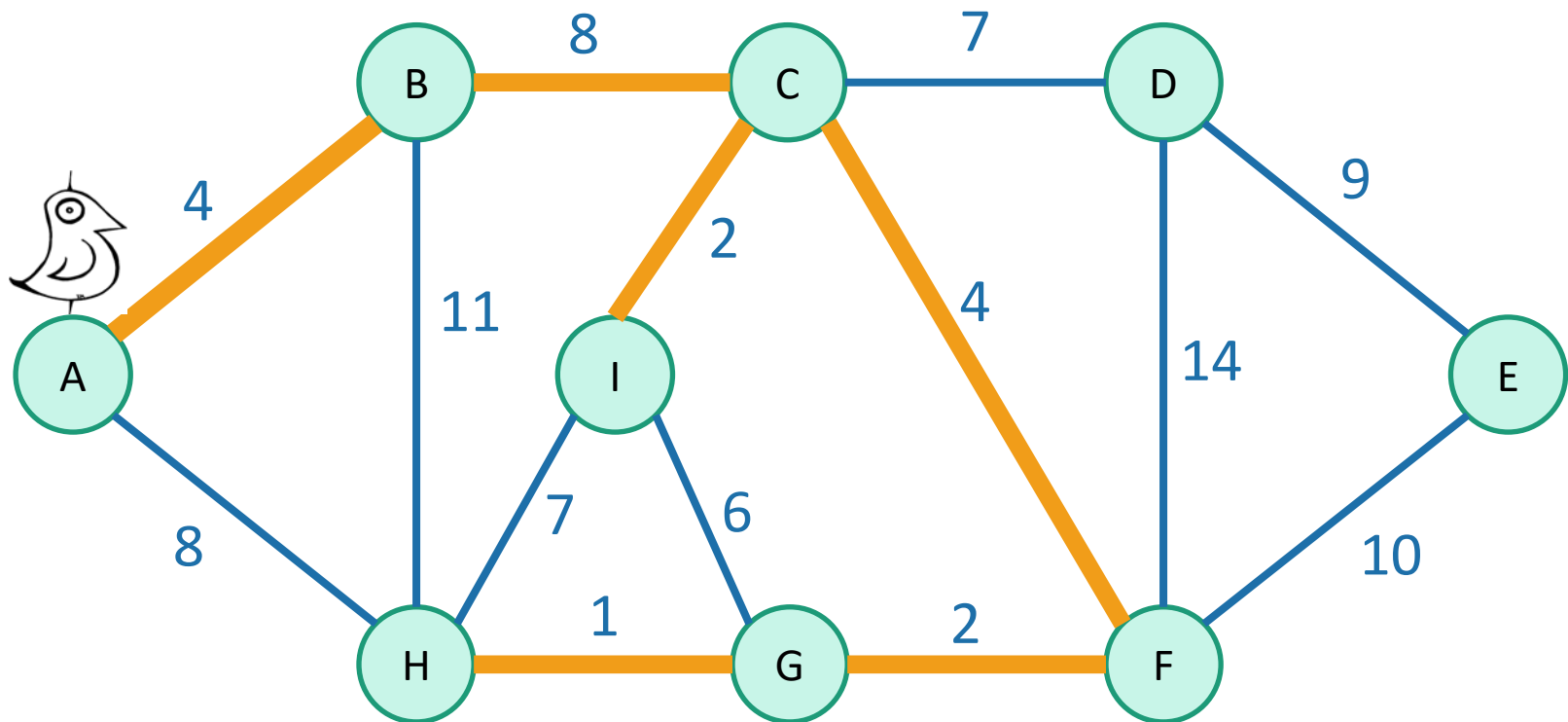
Idea 1

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.
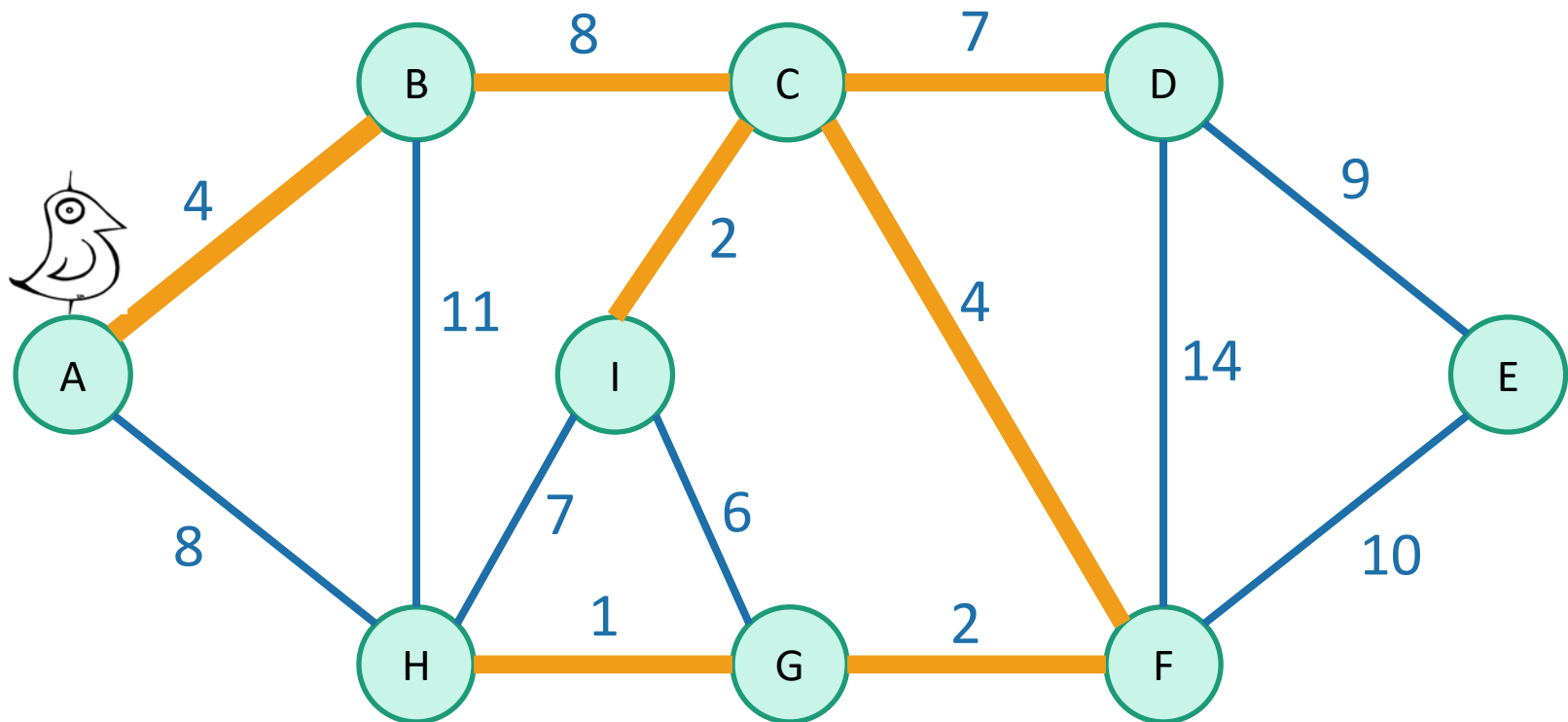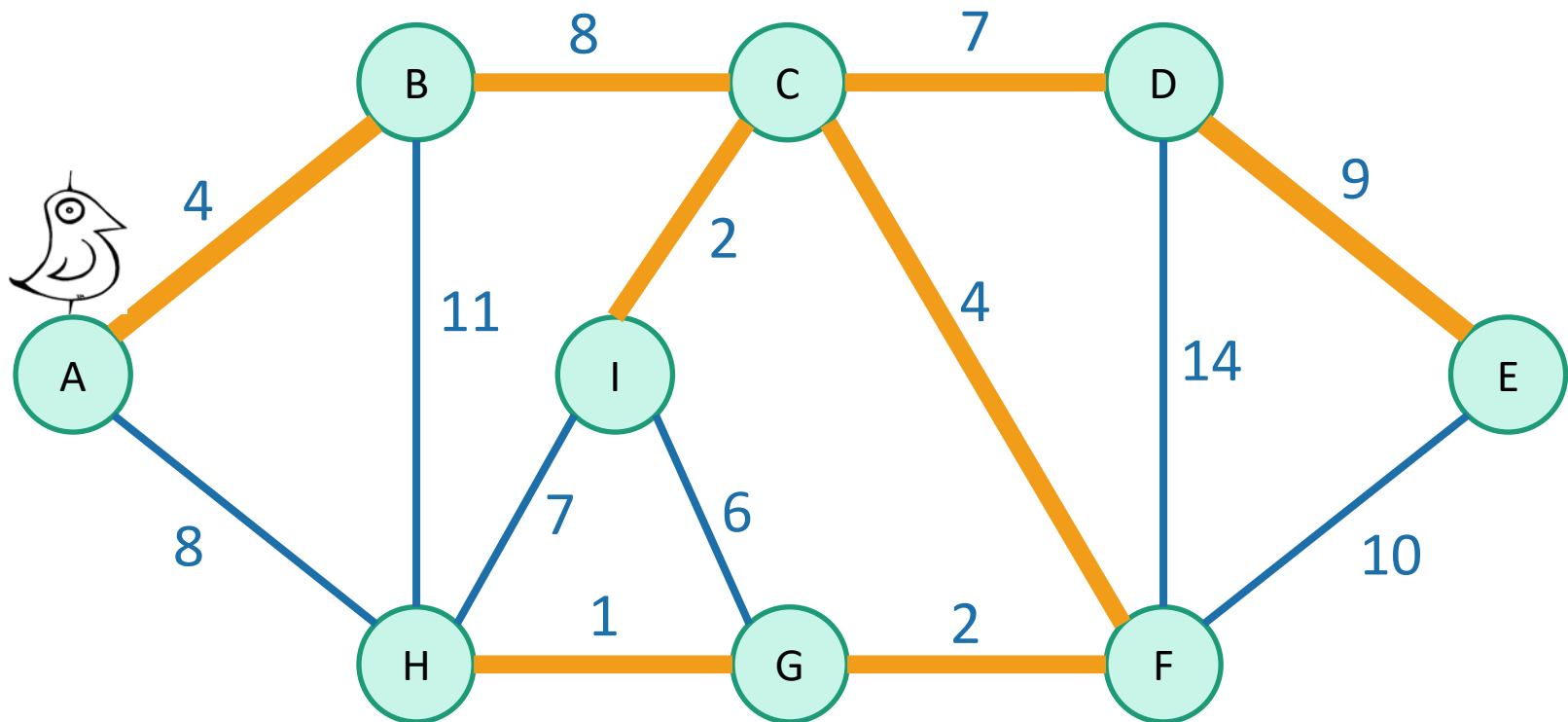
Idea 1

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!

Prim [1957]
Dijkstra [1959]

- 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

At most n iterations of this while loop.

Time at most m to go through all the edges and find the lightest.

Naively, the running time is O(nm):
- For each of ≤n iterations of the while loop:
  - 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:
  - If there exists an MST that contains all of the edges S we have added so far...
  - ...then when we make our next choice {u,v}, there is still an MST containing S and {u,v}.

- 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}}

This edge is light

S is the set of **thick orange** edges
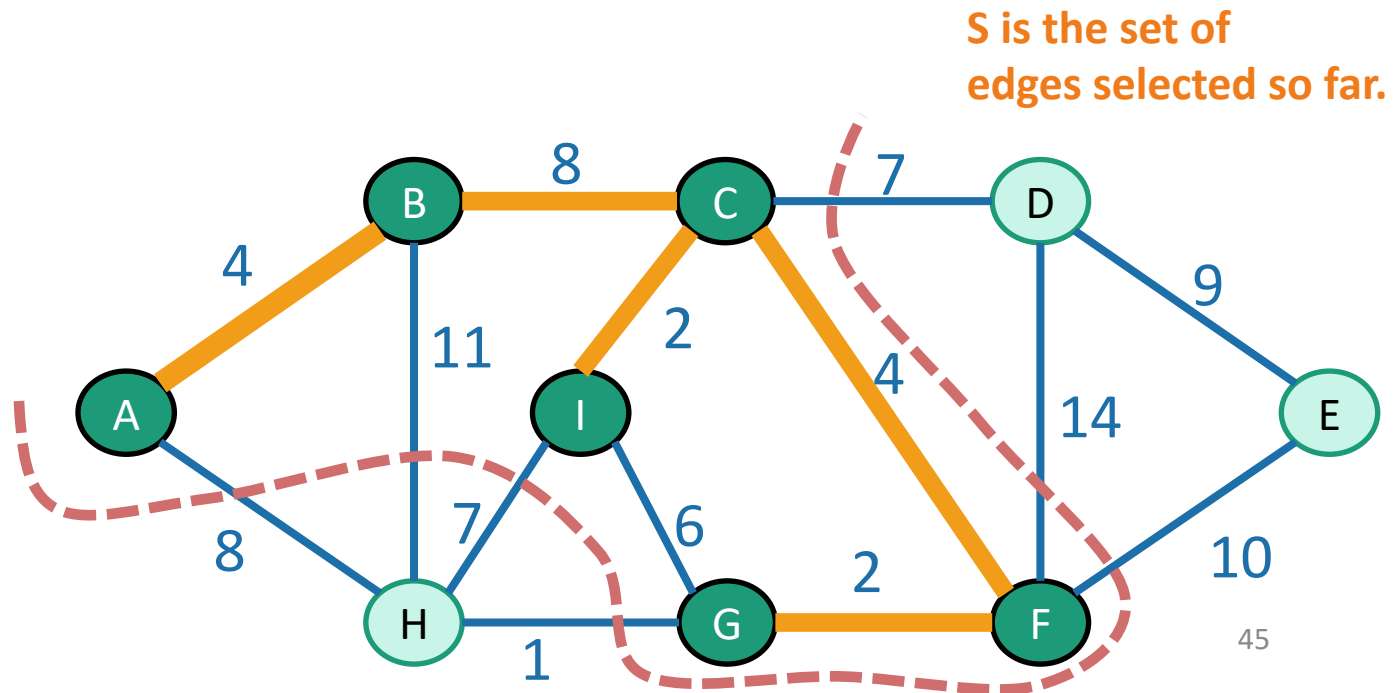
43
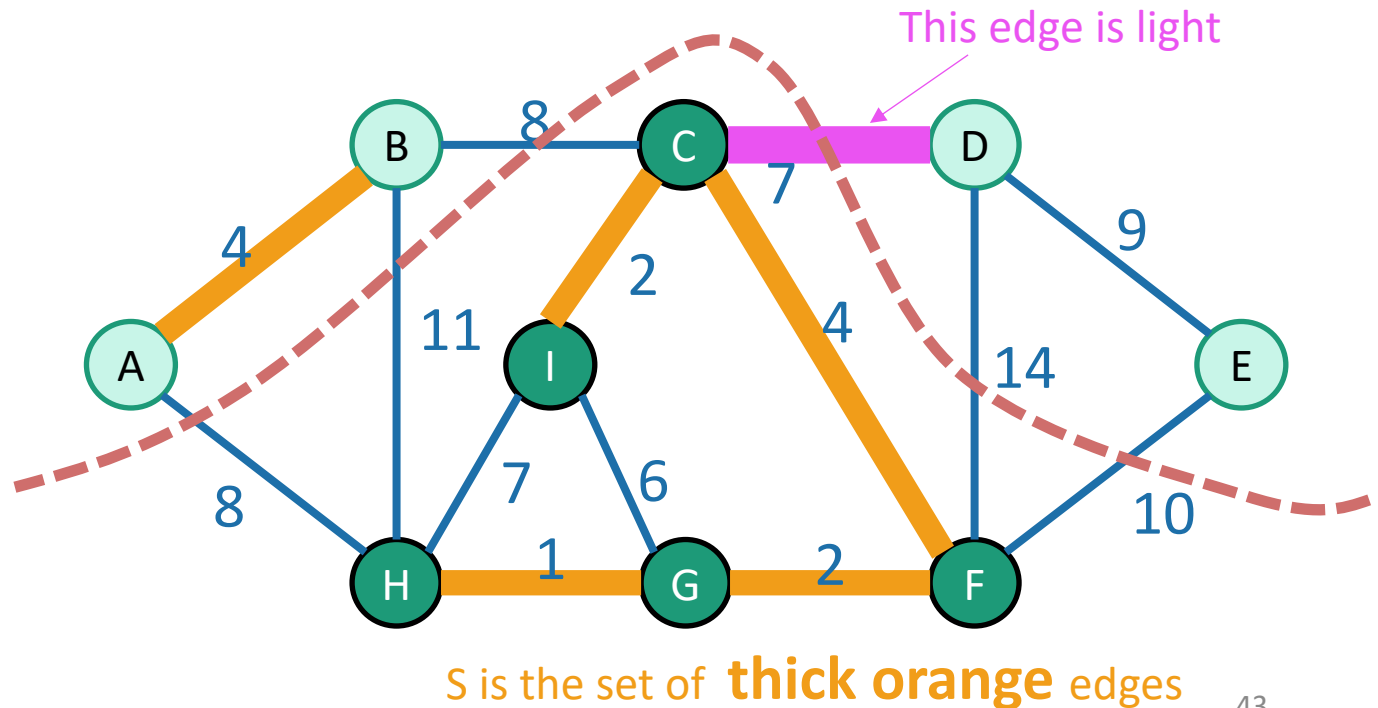
Partway through Prim

- Assume that our choices **S** so far don't rule out success
  - There is an MST consistent with those choices

How can we use our lemma to show that our
next choice also does not rule out success?
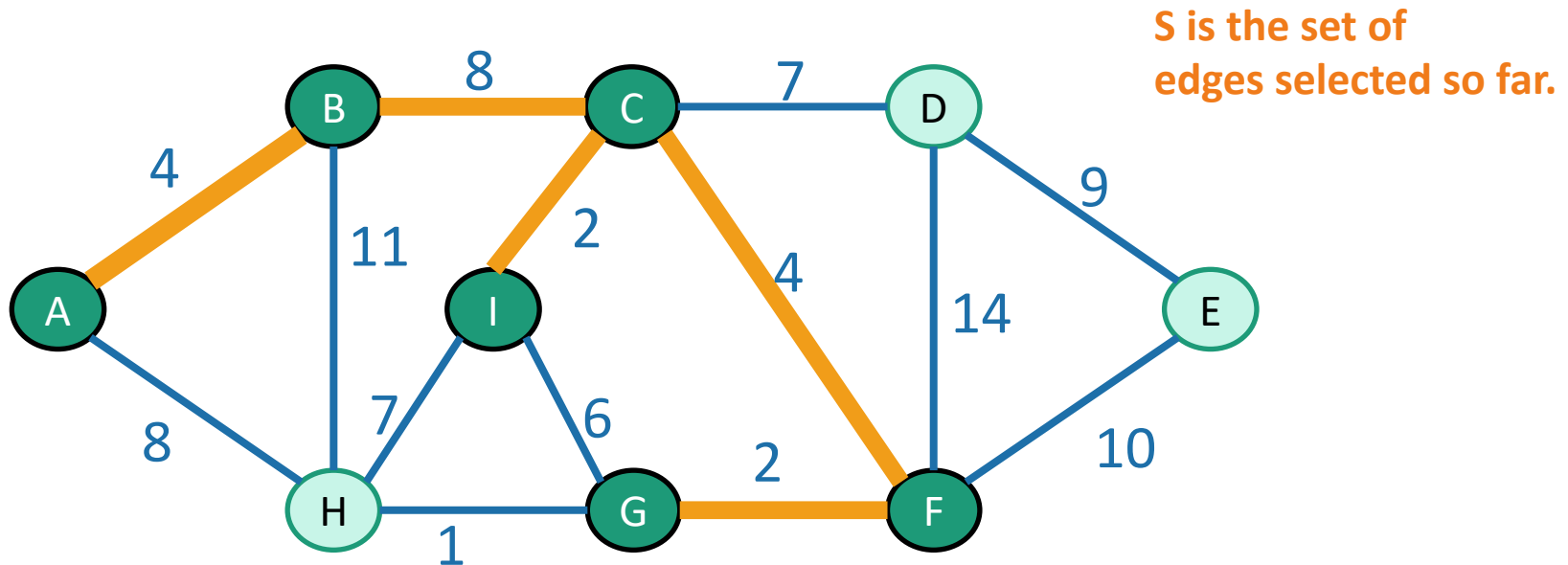
S is the set of
edges selected so far.

Partway through Prim

- Assume that our choices **S** so far don't rule out success
  - There is an MST consistent with those choices
- Consider the cut {**visited**, **unvisited**}
  - This cut respects S.

S is the set of edges selected so far.

Partway through Prim

- Assume that our choices **S** so far don't rule out success
  - There is an MST consistent with these choices

- Consider the cut {**visited**, **unvisited**}
  - This cut respects S.

- The edge we add next is a **light edge**.
  - Least weight of any edge crossing the cut.

- By the Lemma, **that edge** is safe to add.
  - There is still an MST consistent with the new set of edges.
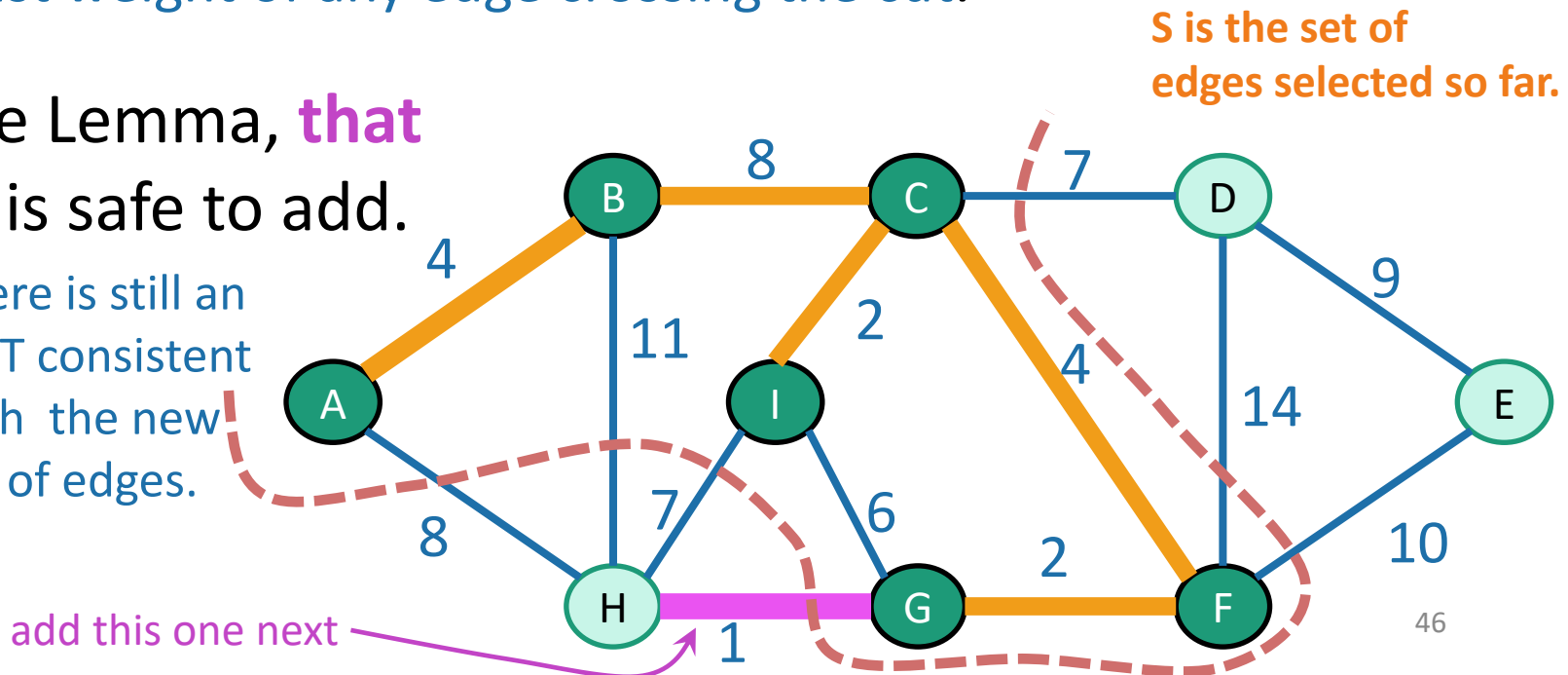
S is the set of edges selected so far.

add this one next

46

So...

- Our greedy choices **don't rule out success**.

- This is enough (along with an argument by induction) to guarantee correctness of Prim's algorithm.

Two questions

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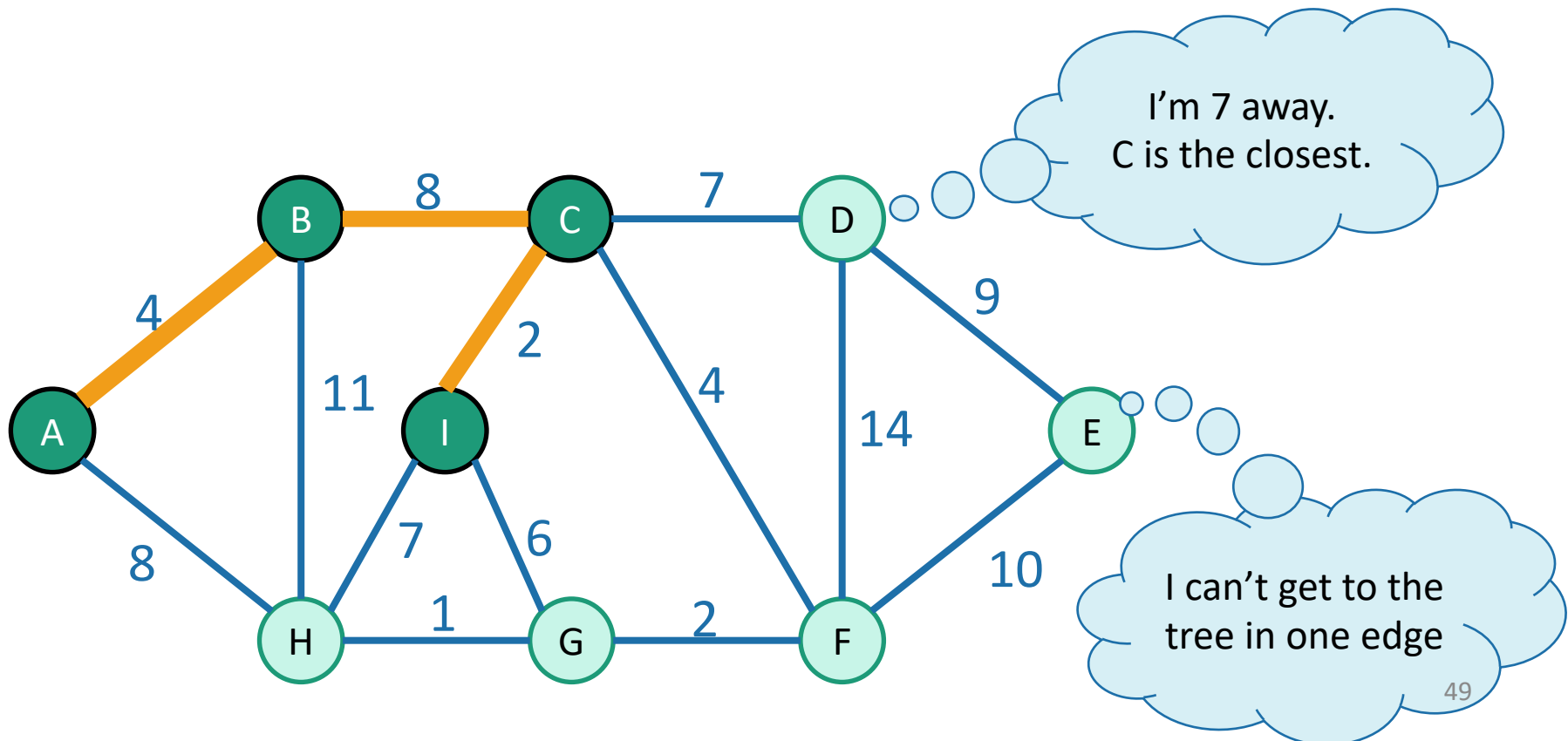   - That is, does it actually return a MST?
     - **Yes!**

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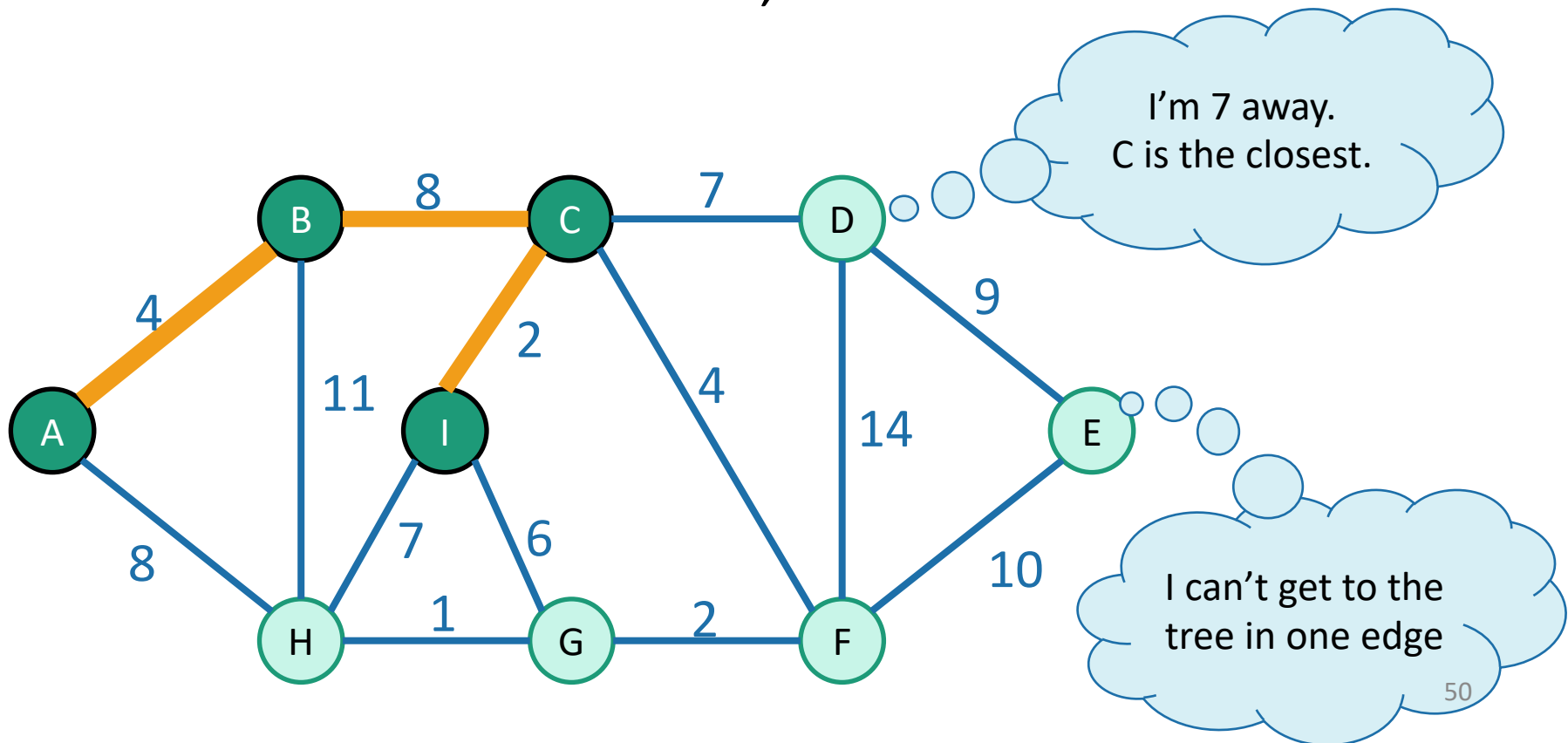   - the pseudocode above says "slowPrim"…
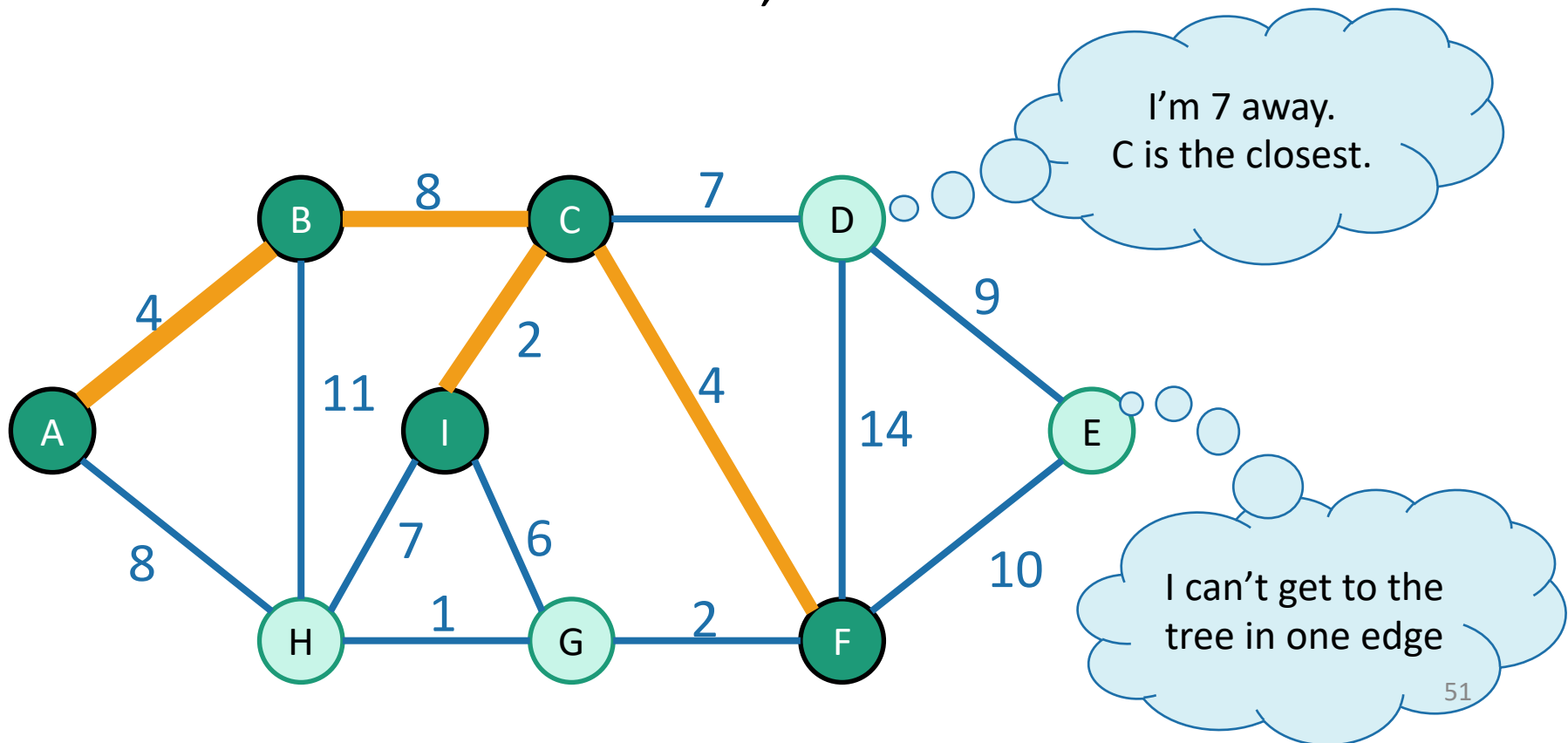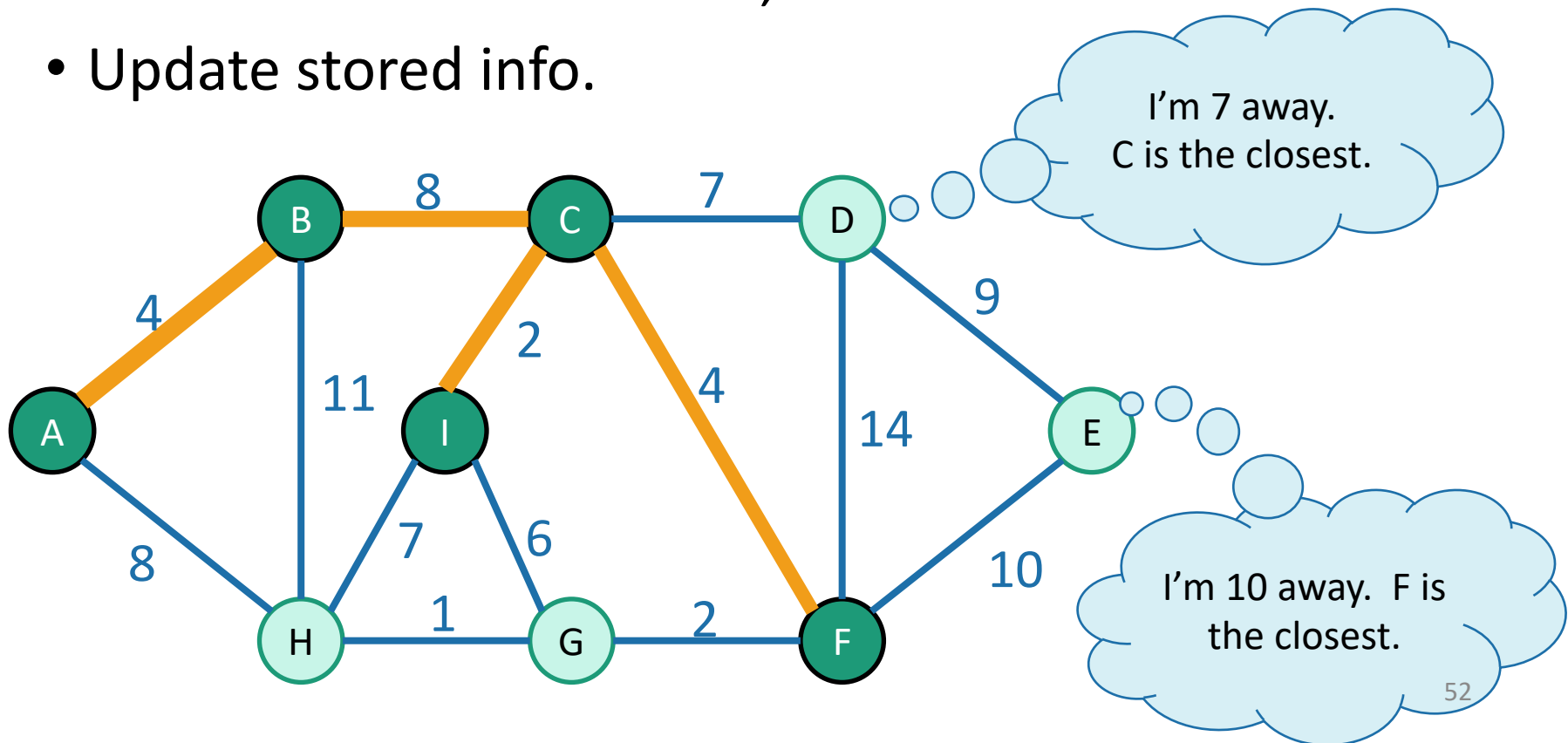
How do we actually implement this?

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

I'm 7 away.
C is the closest.

I can't get to the
tree in one edge
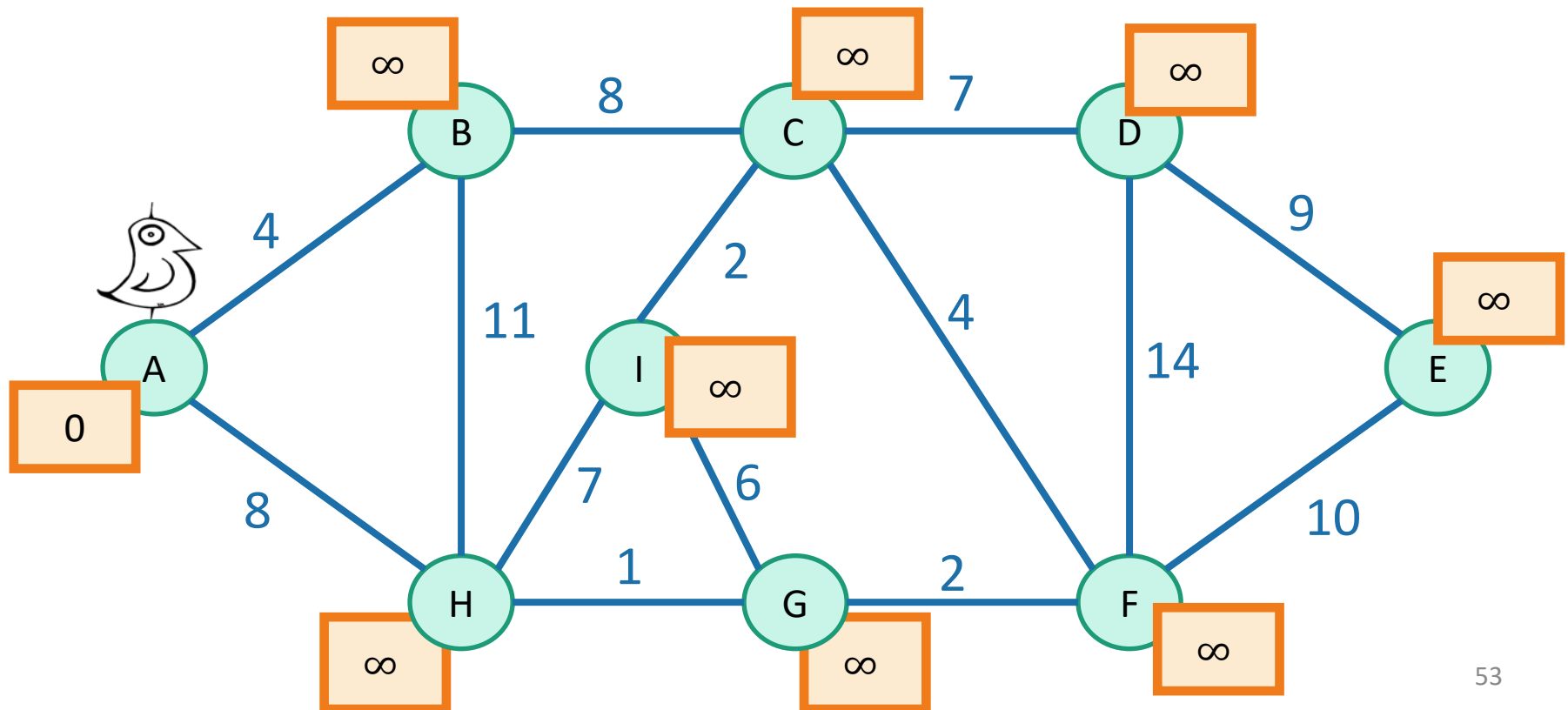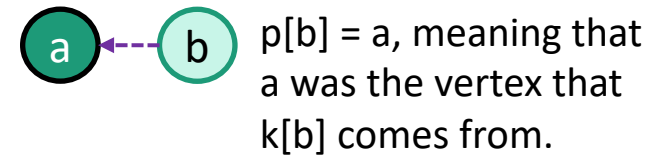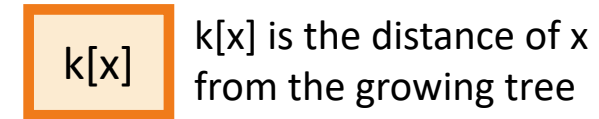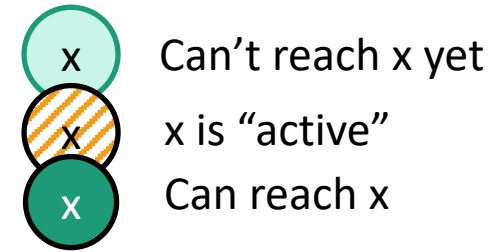
49

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.

I'm 7 away.
C is the closest.

I can't get to the
tree in one edge

50

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.

I'm 7 away.
C is the closest.

I can't get to the tree in one edge

51

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.

I'm 7 away.
C is the closest.

I'm 10 away. F is the closest.

Efficient implementation
Every vertex has a key and a parent
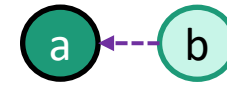
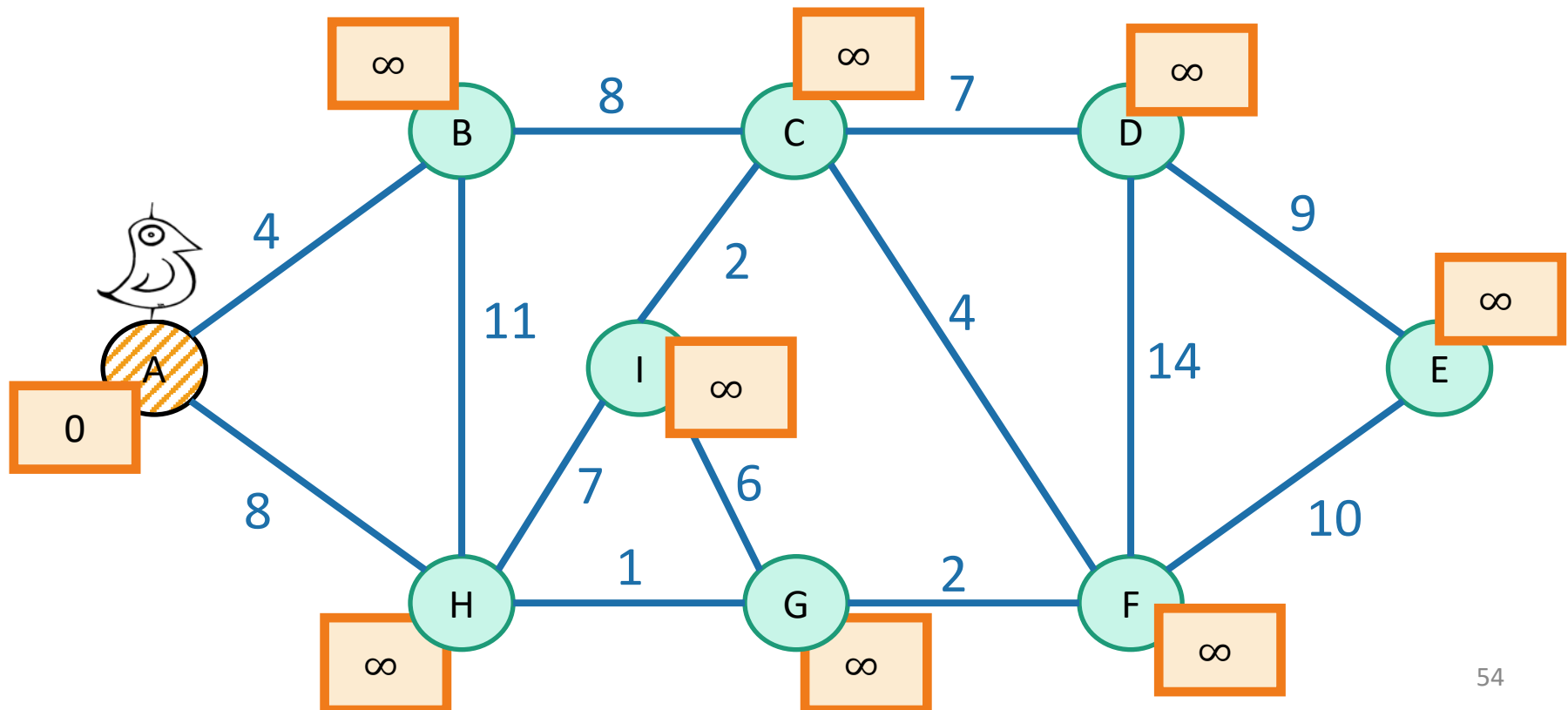**Until** all the vertices are **reached:**

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
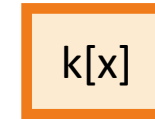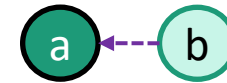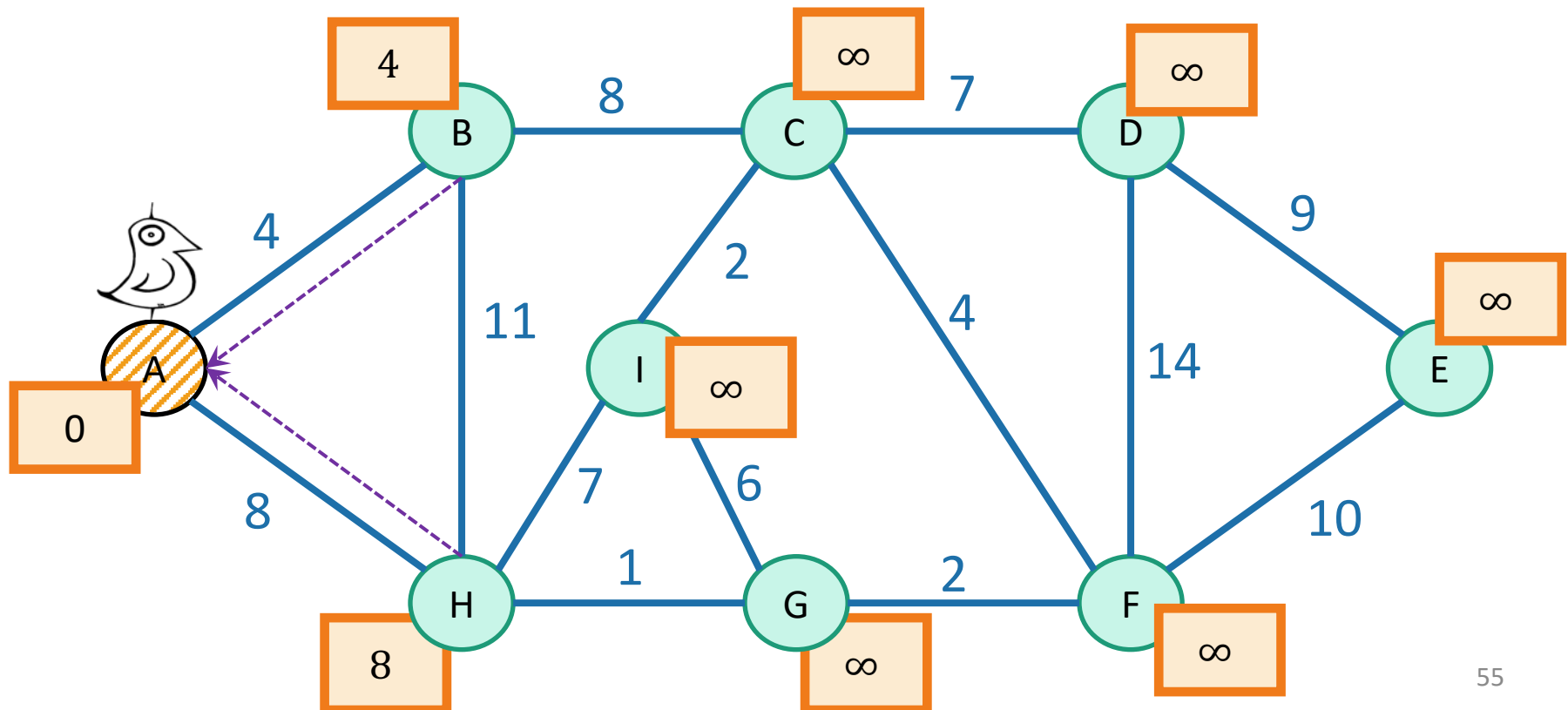
Until all the vertices are **reached:**

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

x   Can't reach x yet

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

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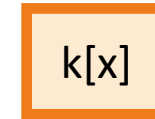
- Activate the **unreached** vertex u with the smallest key.
- **for each** of u's unreached neighbors v:
  - k[v] = min(k[v], weight(u,v))
  - if k[v] updated, p[v] = u

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

Until all the vertices are **reached:**

- Activate the **unreached** vertex u with the smallest key.
- **for each** of u's unreached 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.**

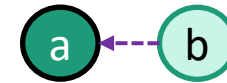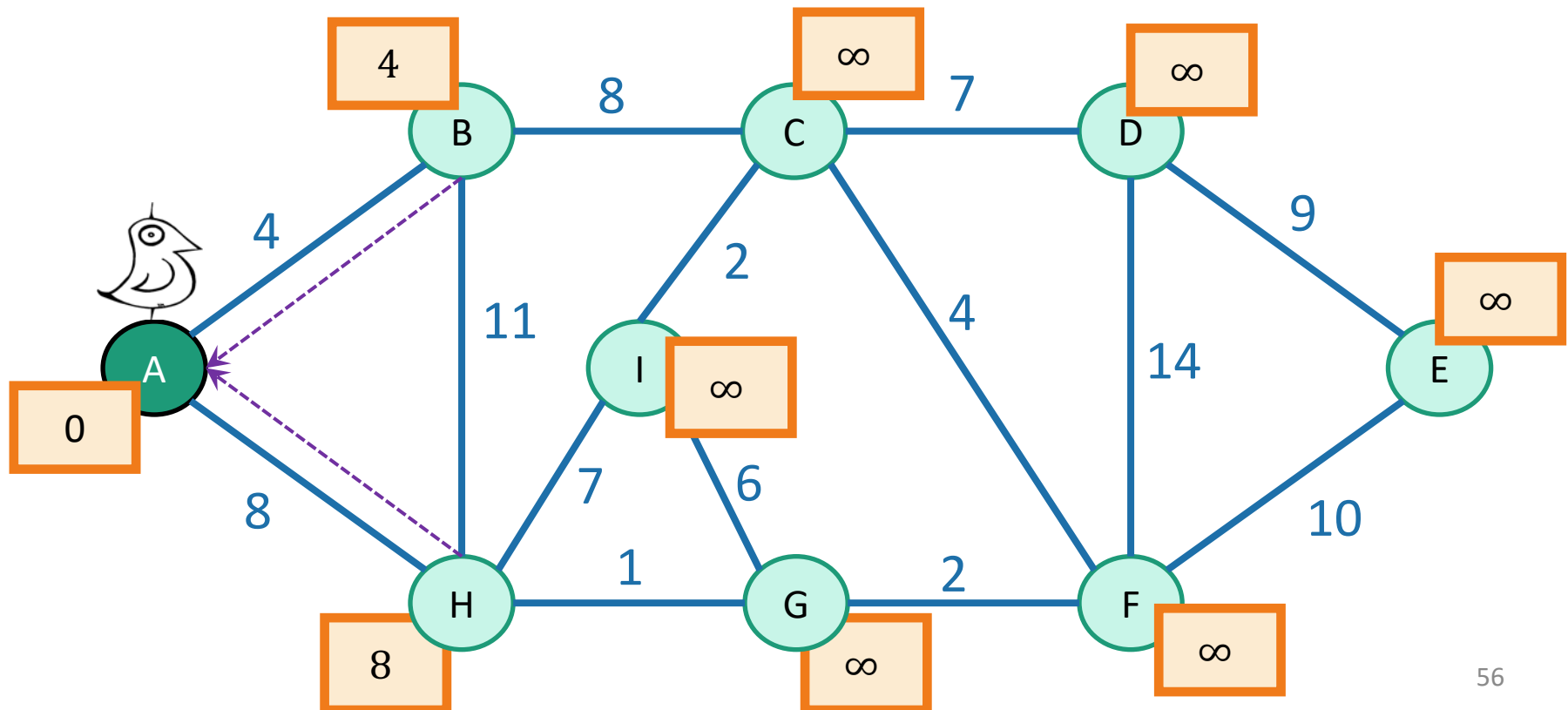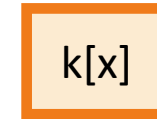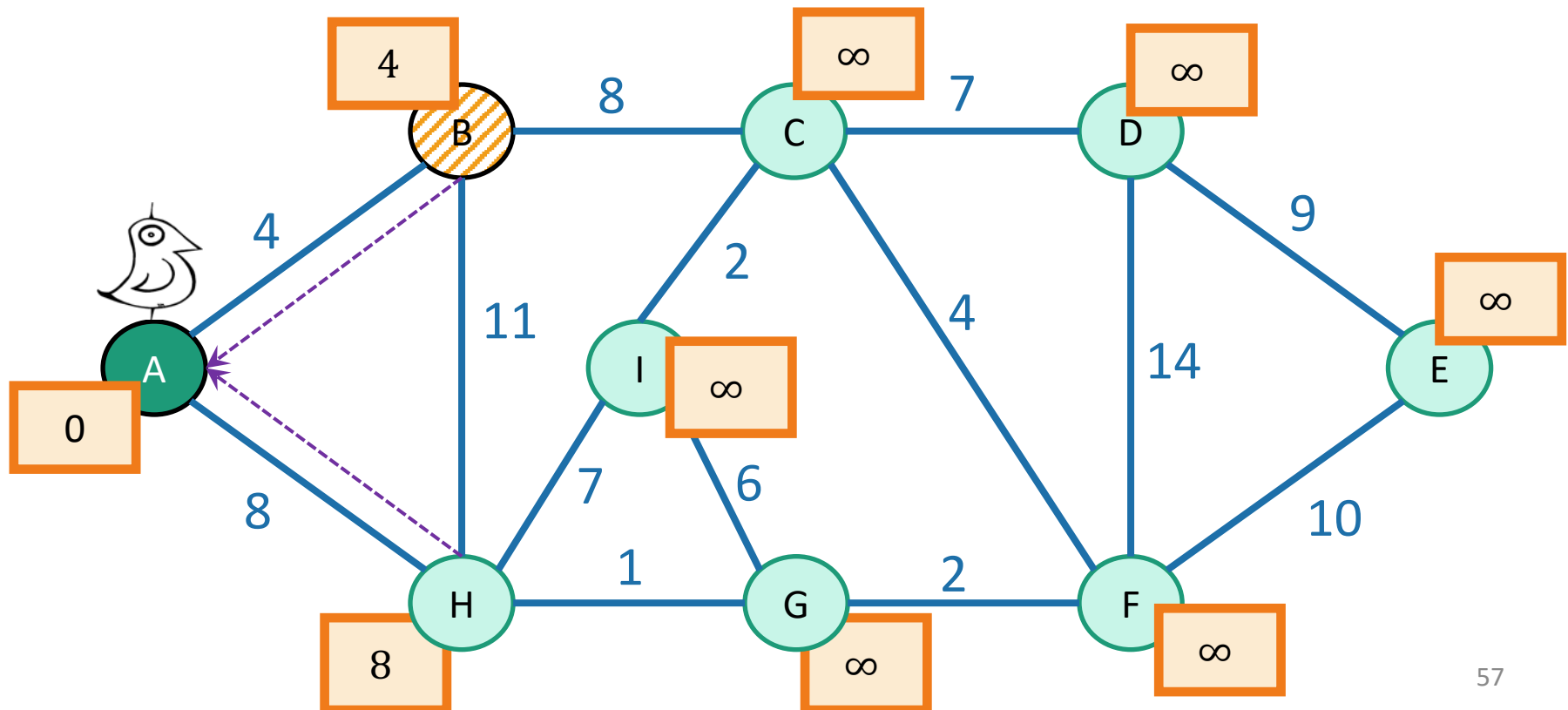x — Can't reach x yet

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

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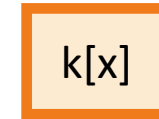
- Activate the **unreached** vertex u with the smallest key.
- **for each** of u's unreached neighbors v:
  - k[v] = min(k[v], weight(u,v))
  - if k[v] updated, p[v] = u
- Mark u as **reached,** and **add (p[u],u) to MST.**

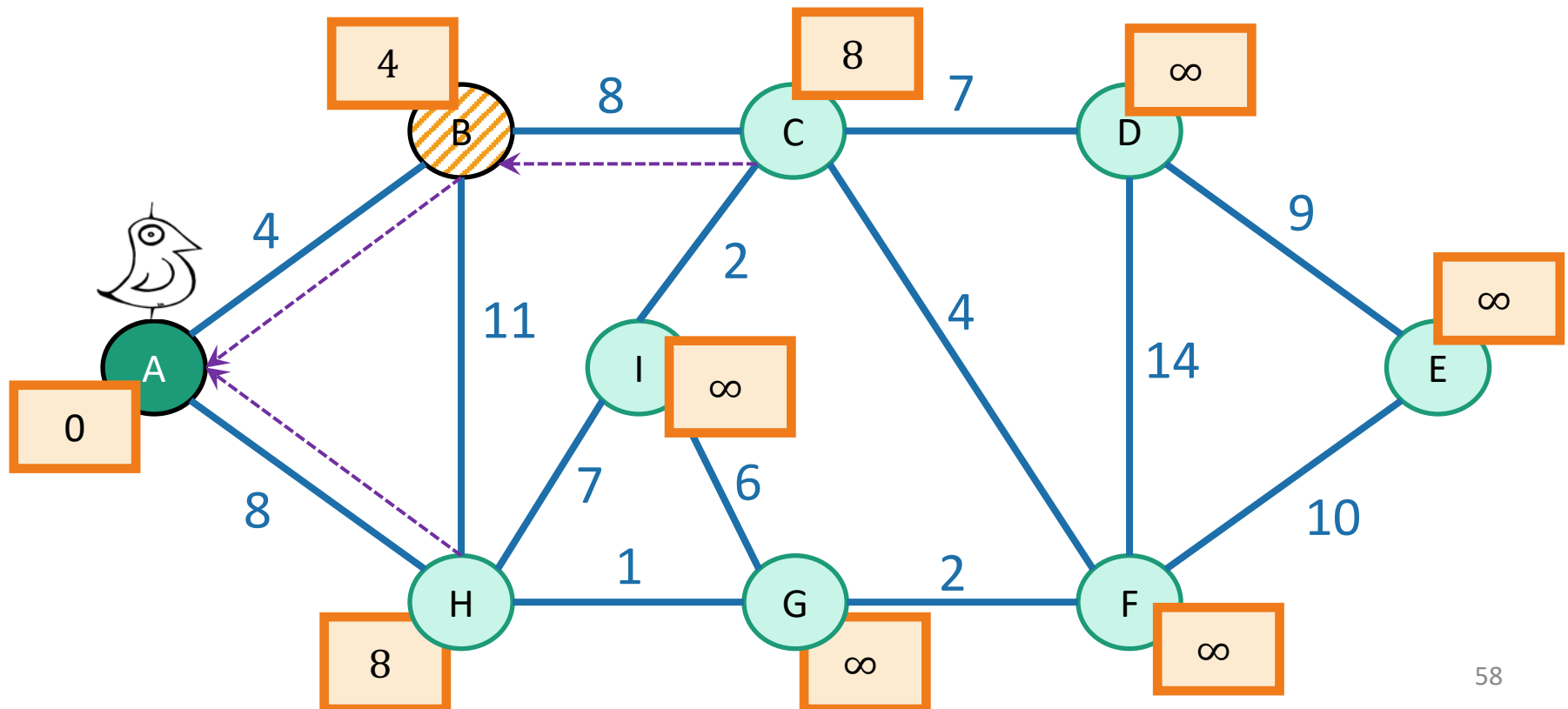x — Can't reach x yet

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

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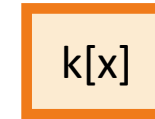
- Activate the **unreached** vertex u with the smallest key.
- **for each** of u's unreached neighbors v:
  - k[v] = min(k[v], weight(u,v))
  - if k[v] updated, p[v] = u
- Mark u as **reached,** and **add (p[u],u) to MST.**

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

Until all the vertices are **reached:**

- Activate the **unreached** vertex u with the smallest key.
- **for each** of u's unreached 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.**

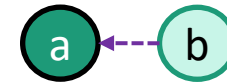x   Can't reach x yet

x   x is "active"
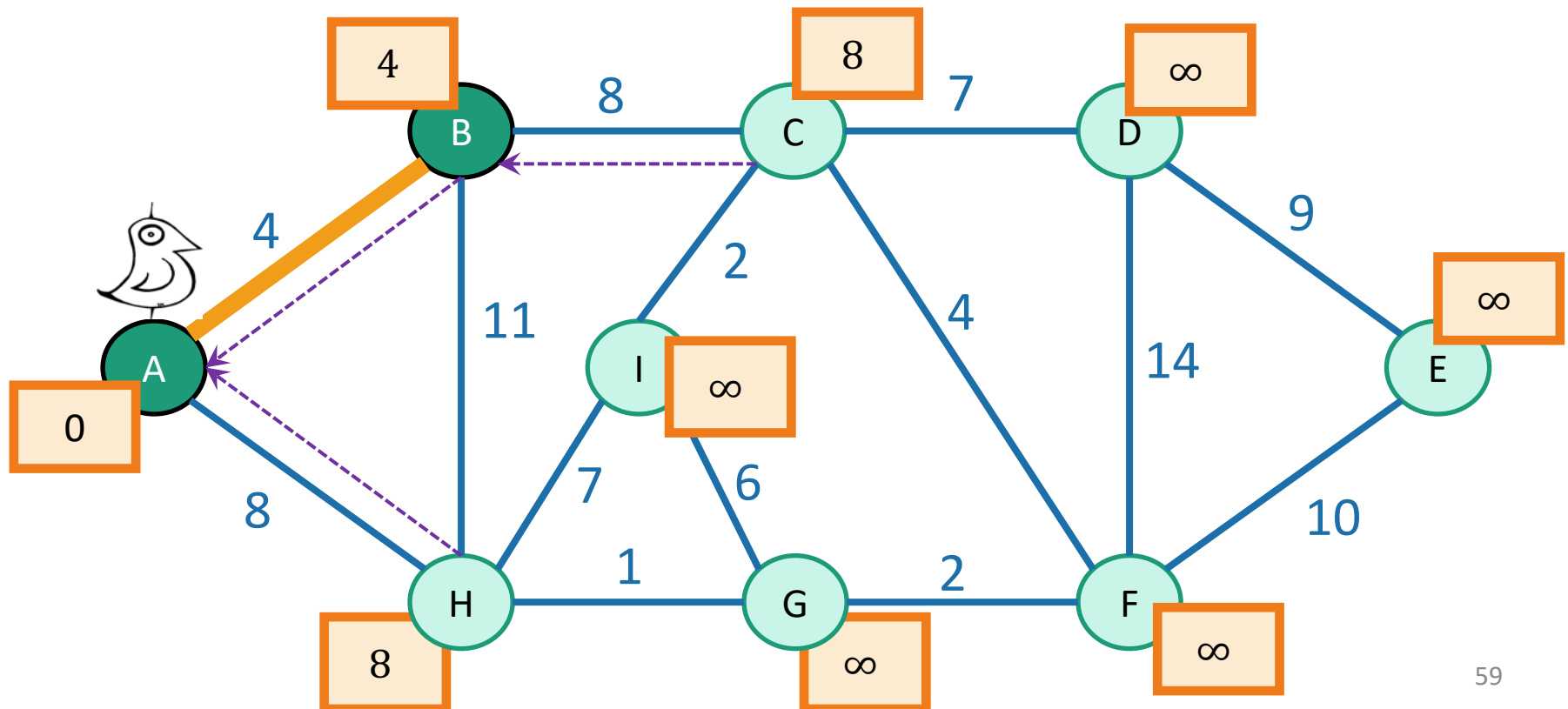
x   Can reach x

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.

59

Efficient implementation
Every vertex has a key and a parent

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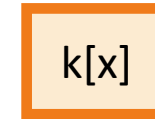
- Activate the **unreached** vertex u with the smallest key.
- **for each** of u's unreached neighbors v:
  - k[v] = min(k[v], weight(u,v))
  - if k[v] updated, p[v] = u
- Mark u as **reached,** and **add (p[u],u) to MST.**

x — Can't reach x yet
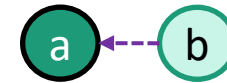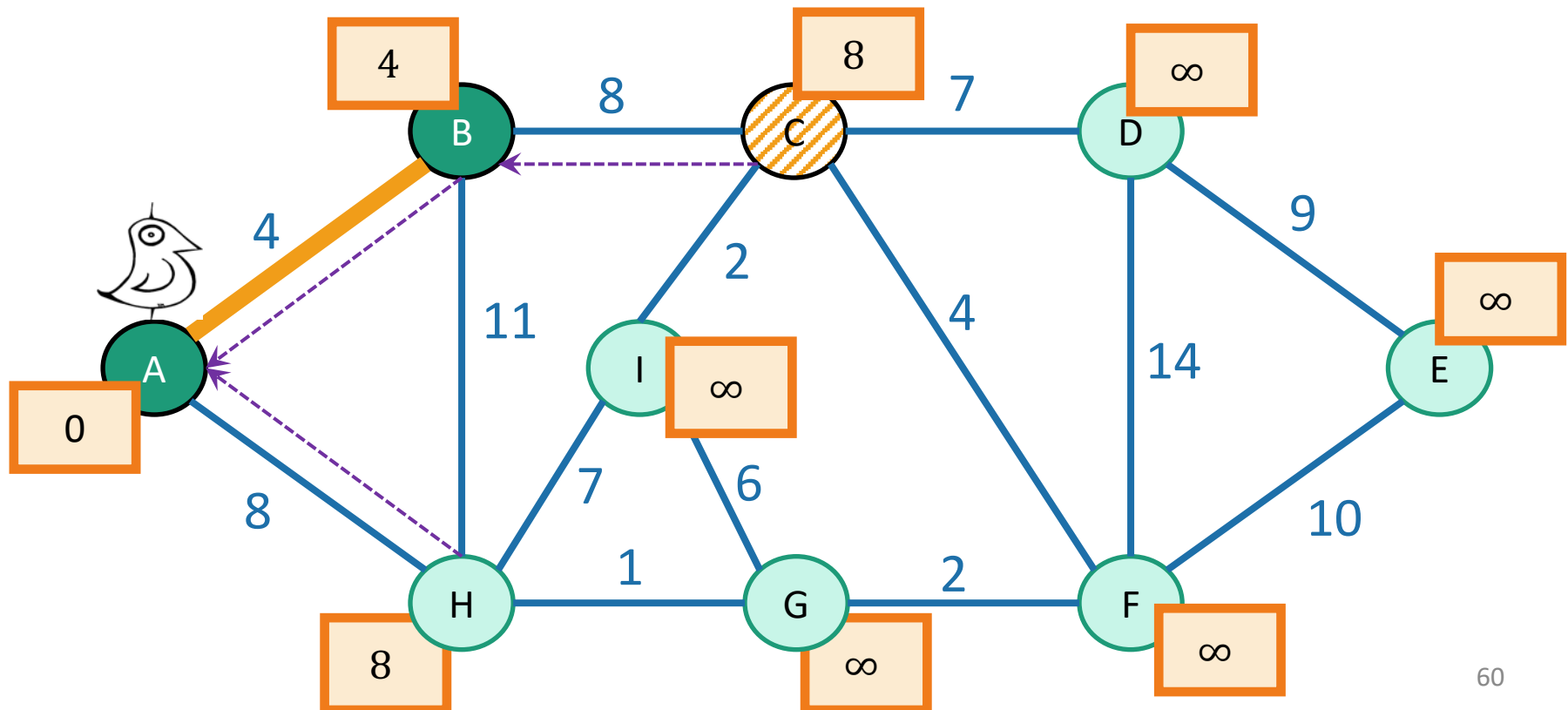
x — x is "active"

x — Can reach x

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

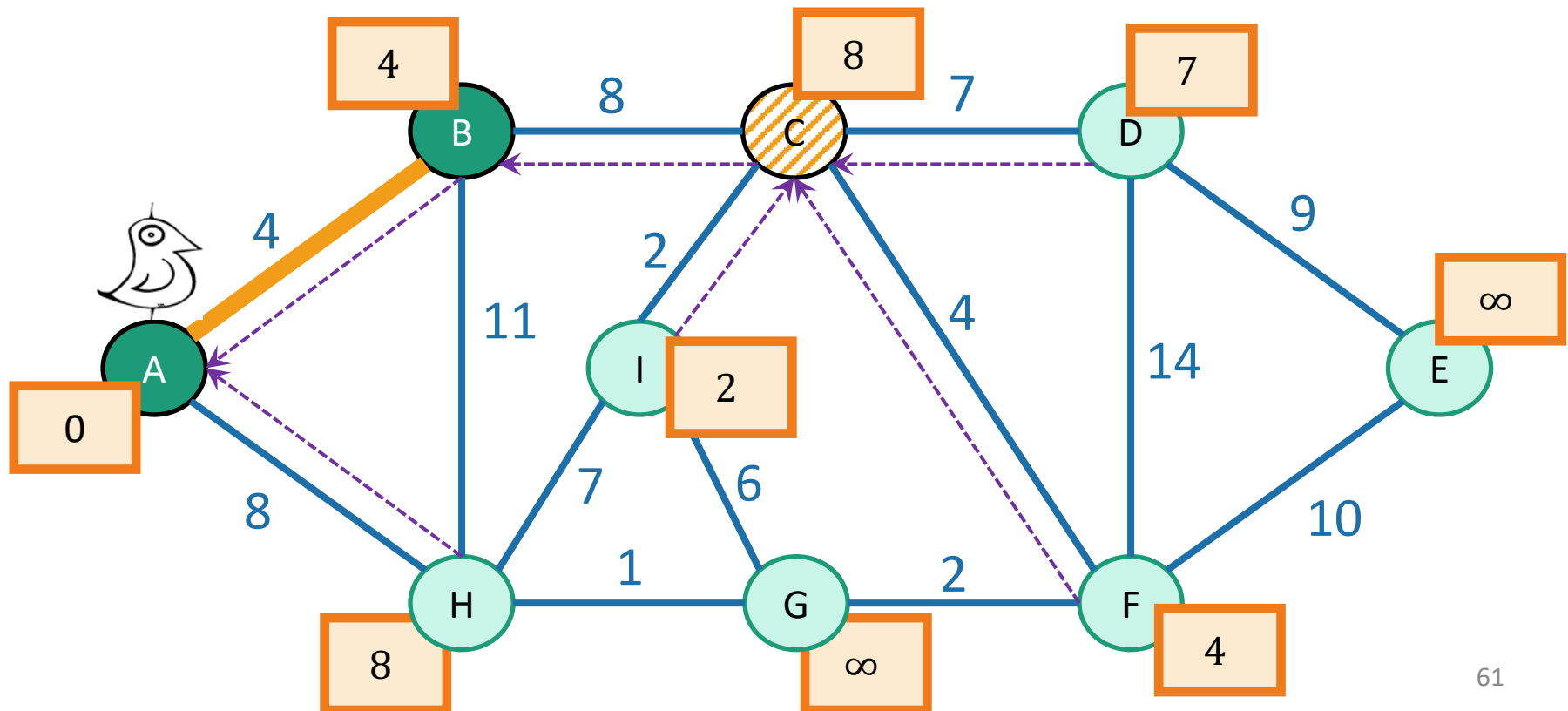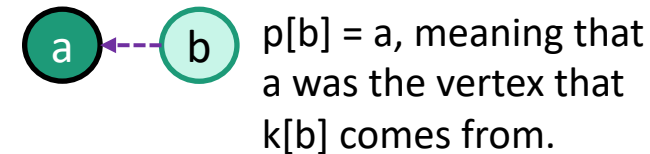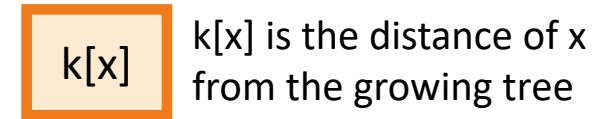a ⟵ b — p[b] = a, meaning that a was the vertex that k[b] comes from.

Efficient implementation

Every vertex has a key and a parent

Until all the vertices are **reached:**

- Activate the **unreached** vertex u with the smallest key.
- **for each** of u's unreached 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 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.
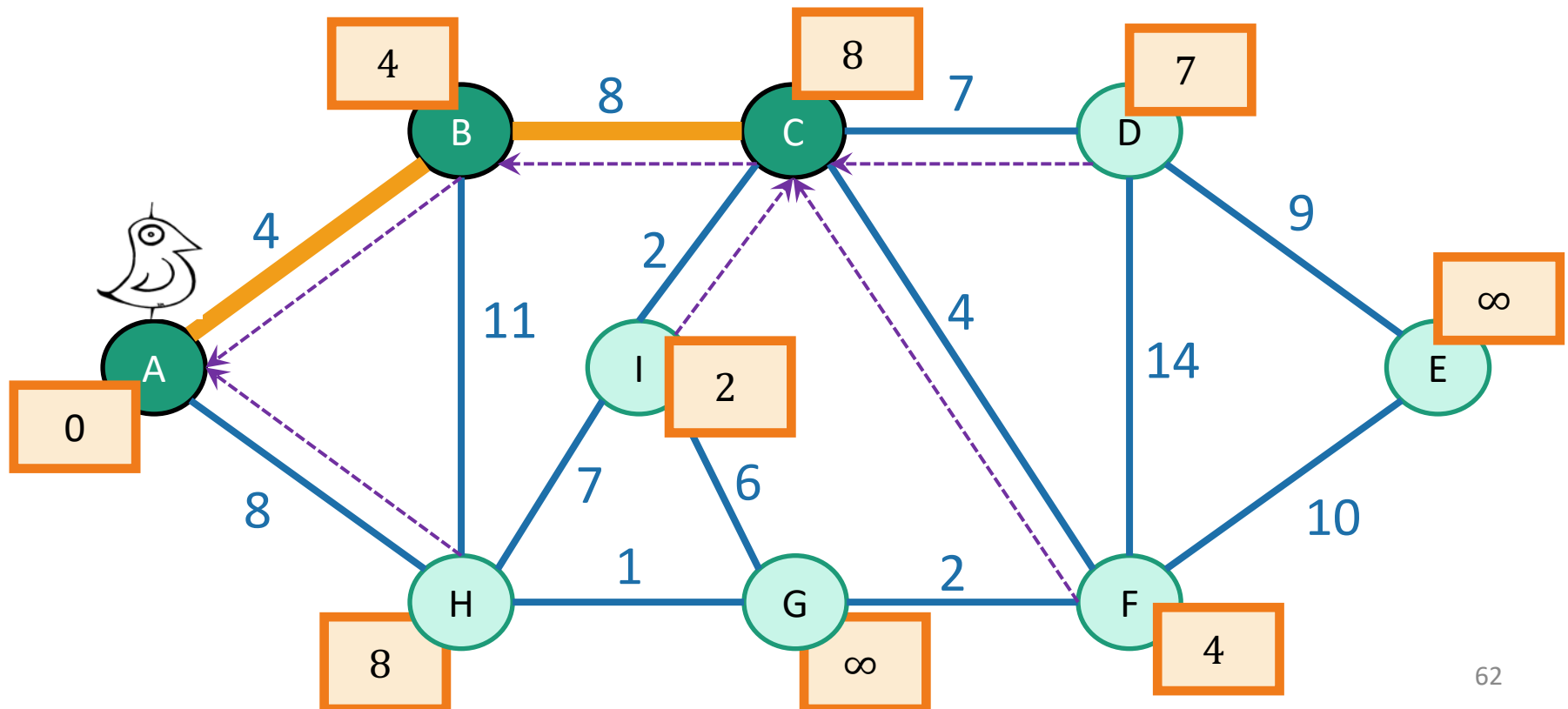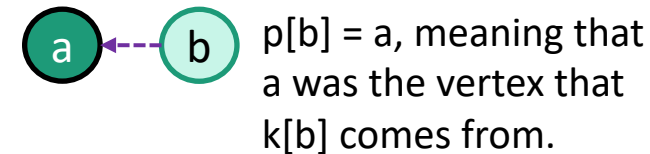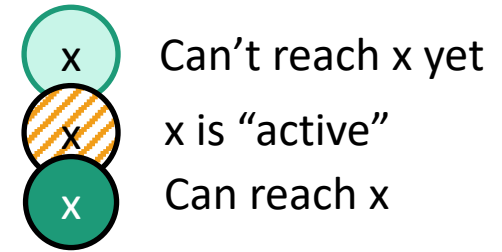
61

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 unreached 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.
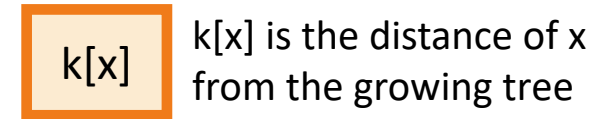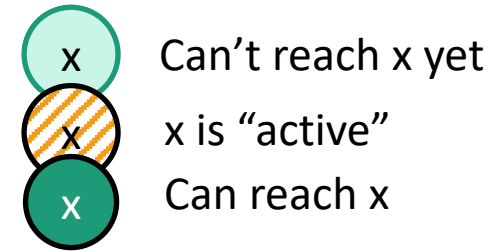
Efficient implementation

Every vertex has a key and a parent

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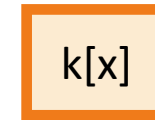
- Activate the **unreached** vertex u with the smallest key.
- **for each** of u's unreached neighbors v:
    - k[v] = min(k[v], weight(u,v))
    - if k[v] updated, p[v] = u
- Mark u as **reached,** and **add (p[u],u) to MST.**

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.

63

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 unreached 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.**
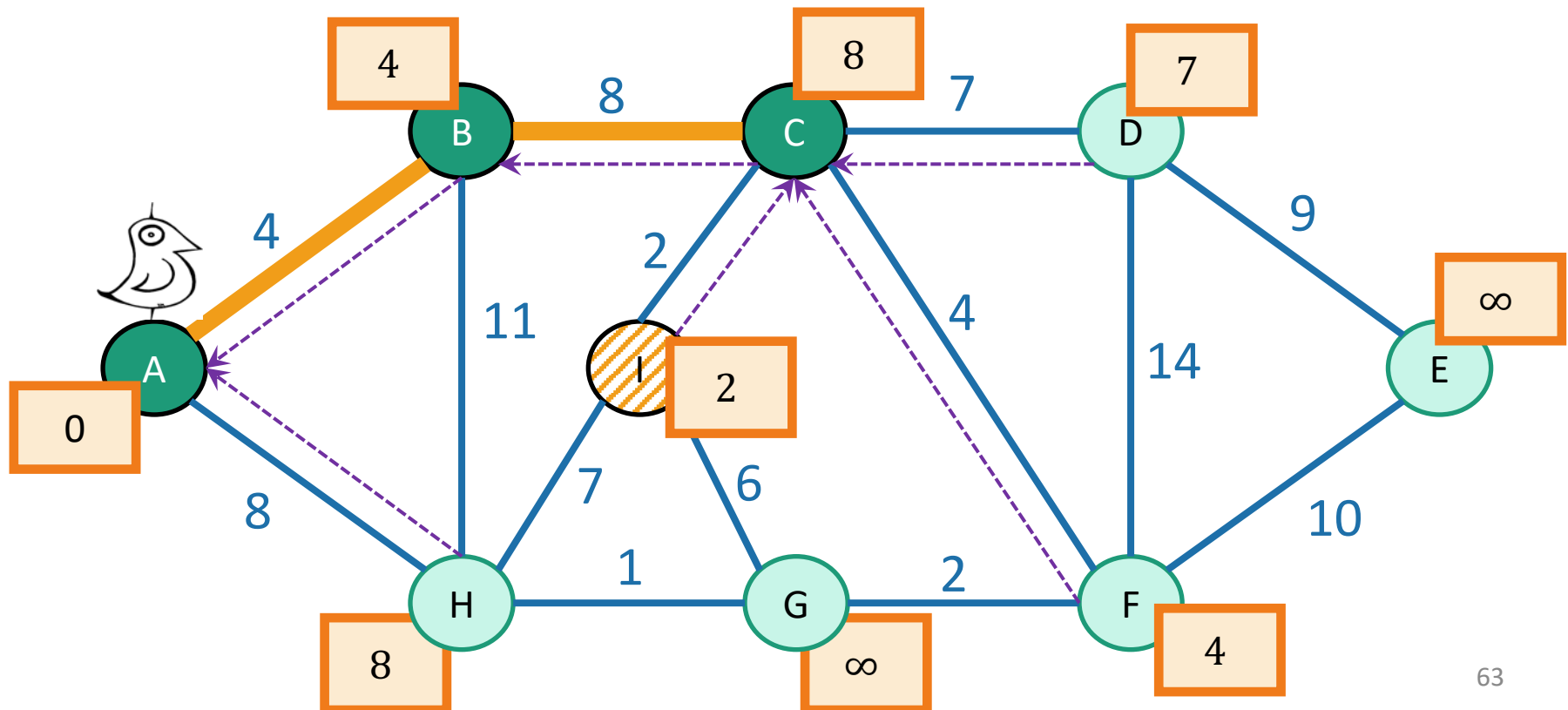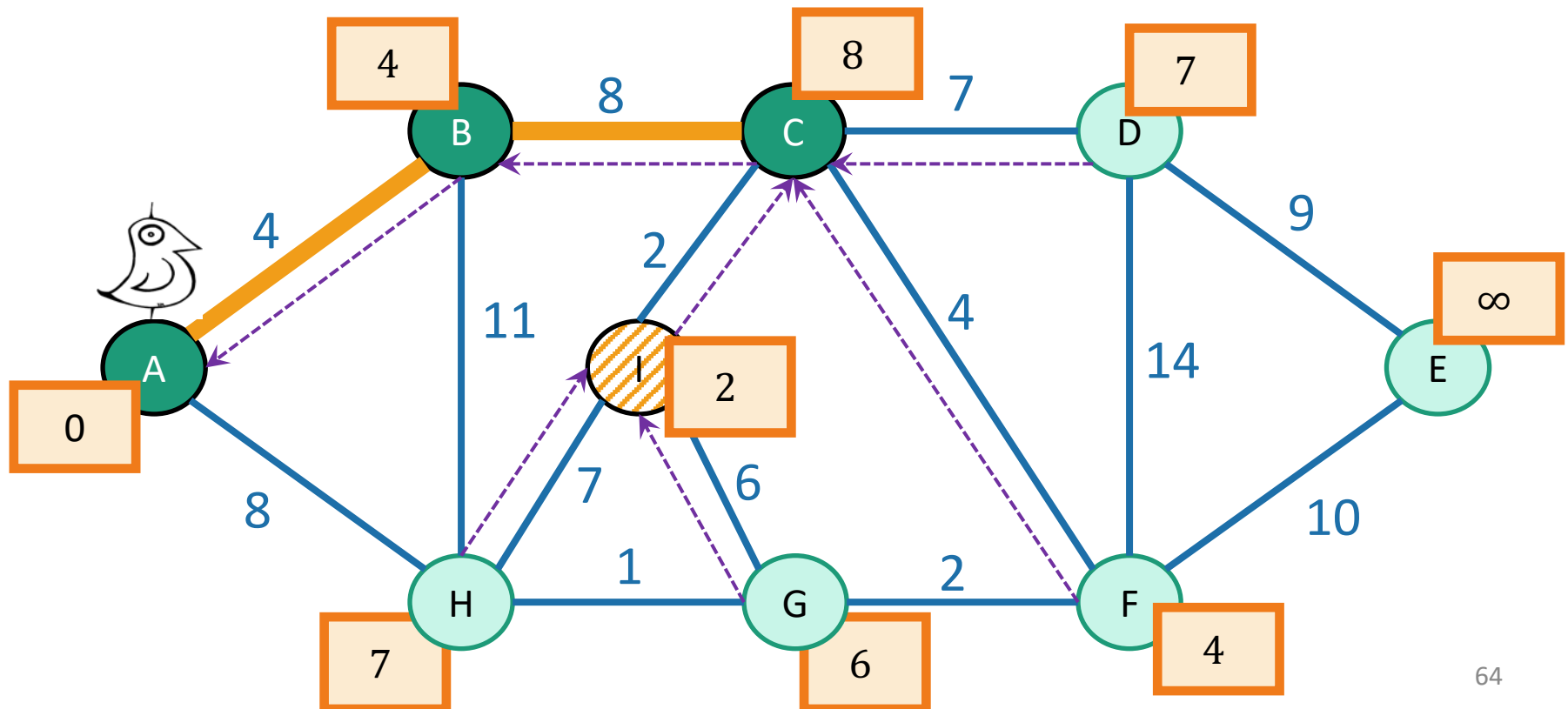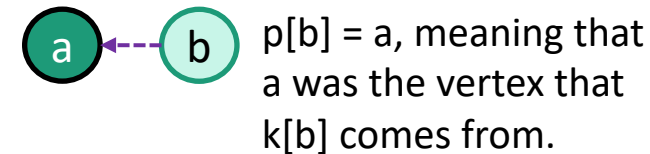

x Can't reach x yet

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.
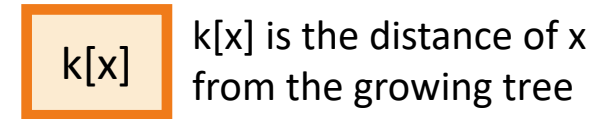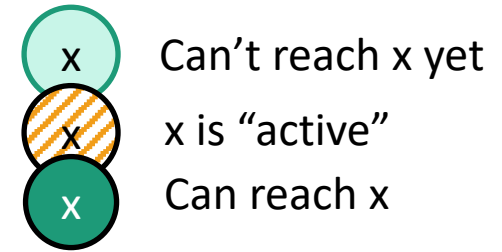

64

Efficient implementation
Every vertex has a key and a parent

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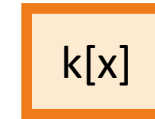
- Activate the **unreached** vertex u with the smallest key.
- **for each** of u's unreached neighbors v:
    - k[v] = min(k[v], weight(u,v))
    - if k[v] updated, p[v] = u
- Mark u as **reached,** and **add (p[u],u) to MST.**

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

Until all the vertices are **reached:**

- Activate the **unreached** vertex u with the smallest key.
- **for each** of u's unreached 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"
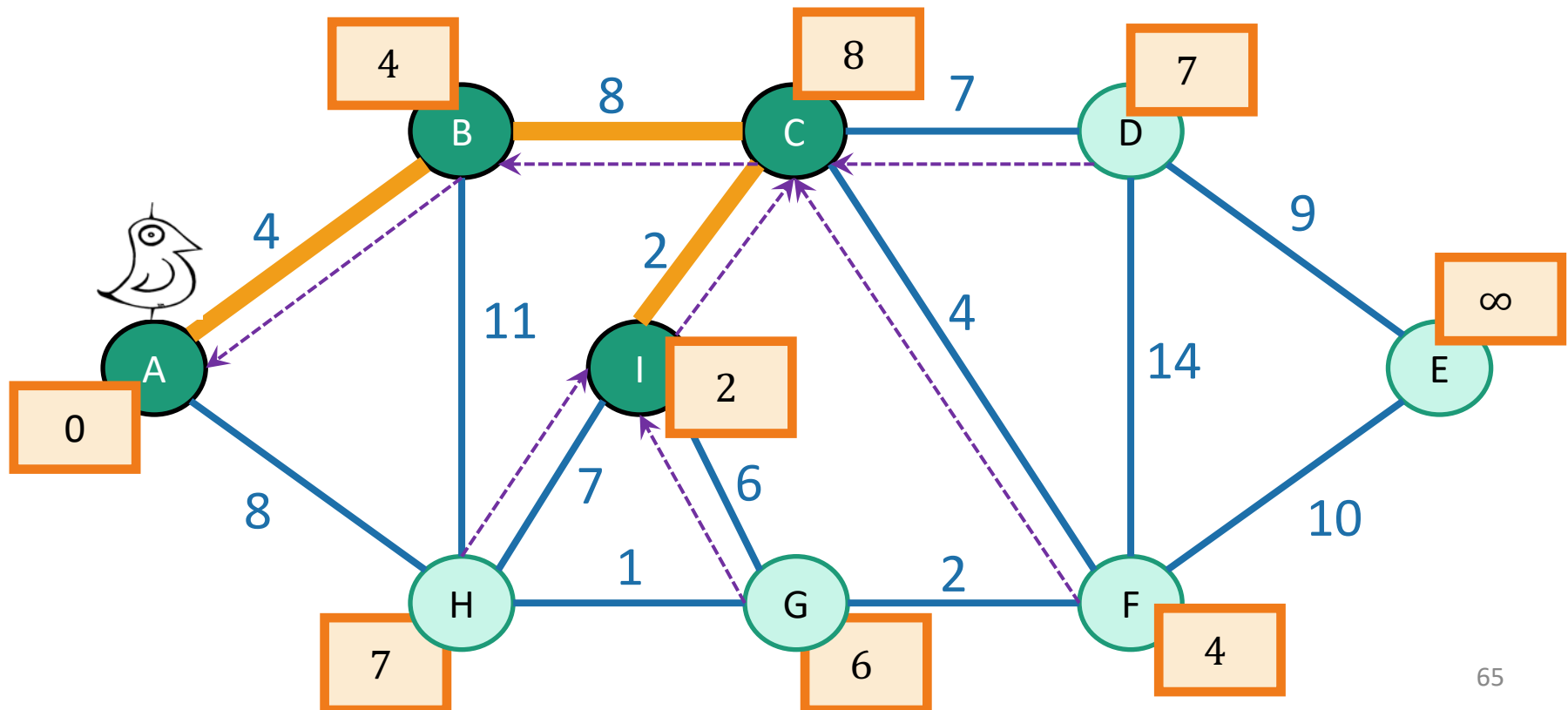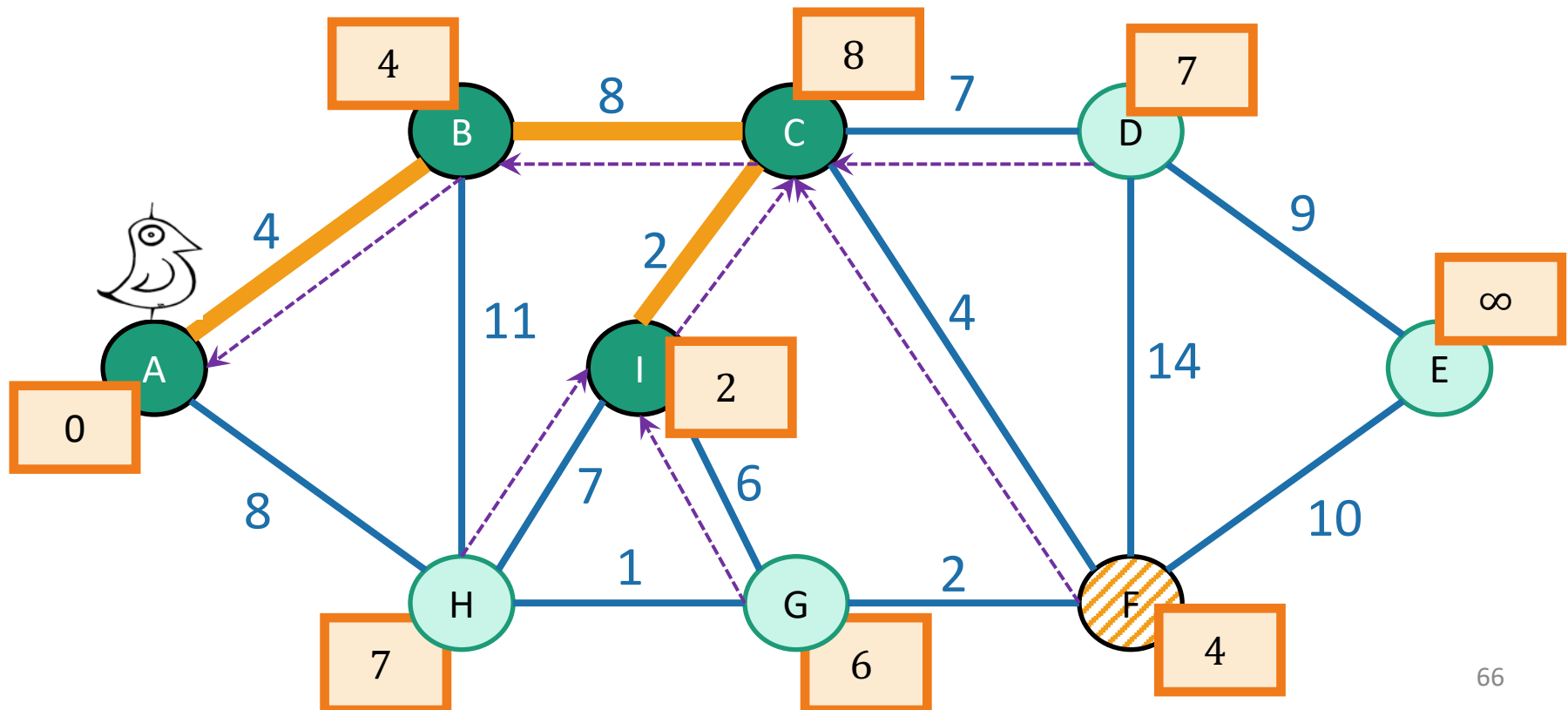
x — Can reach x

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.
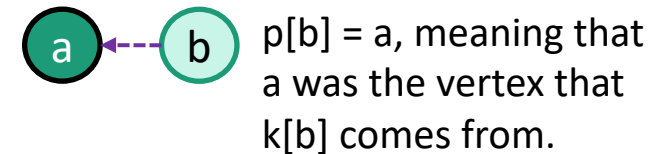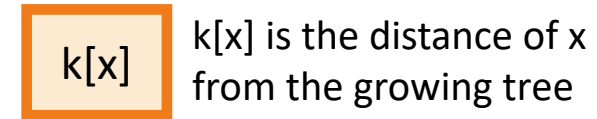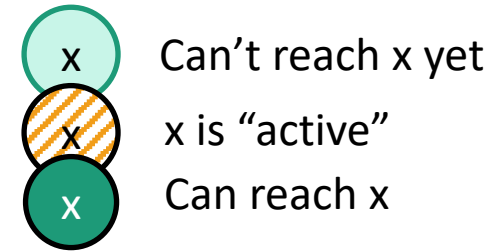

66

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 unreached 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
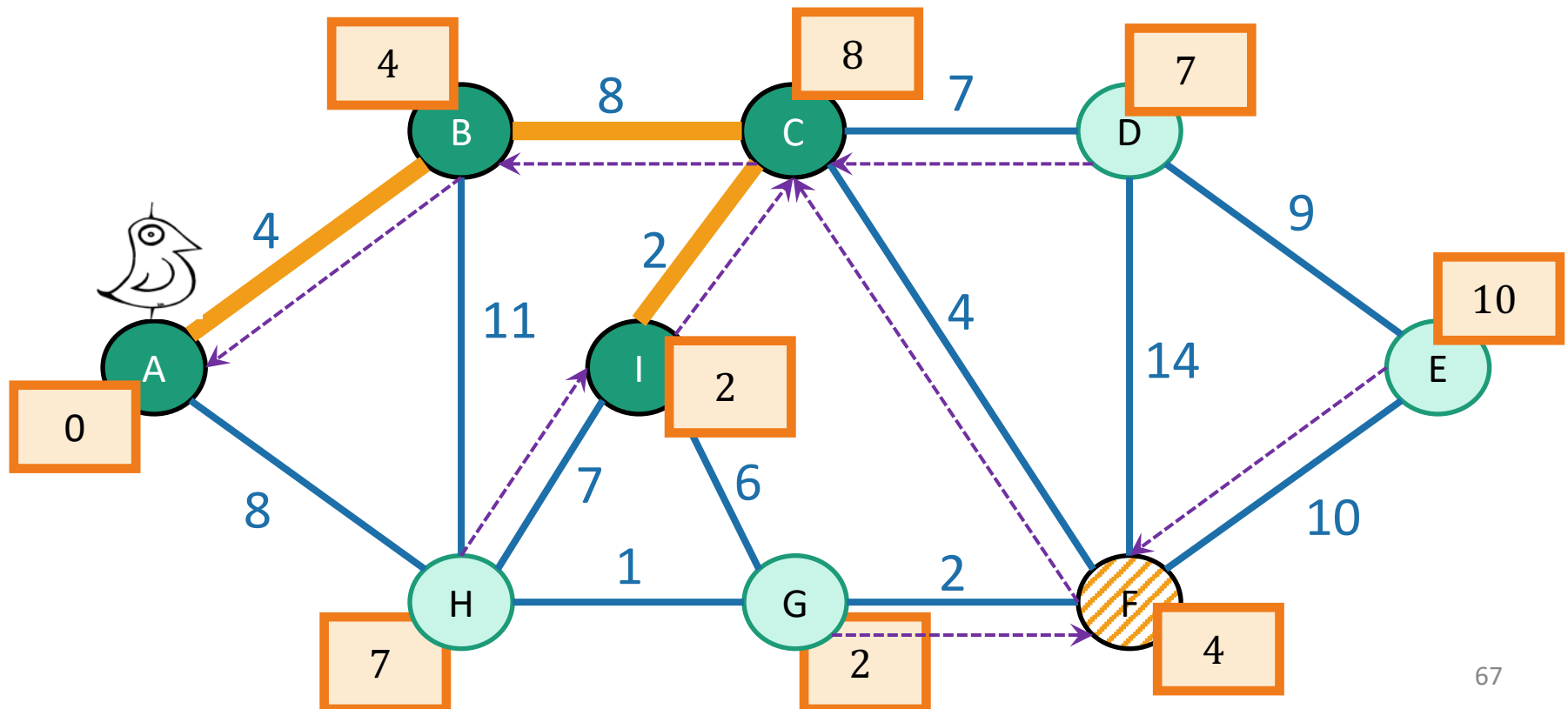
x is "active"

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.
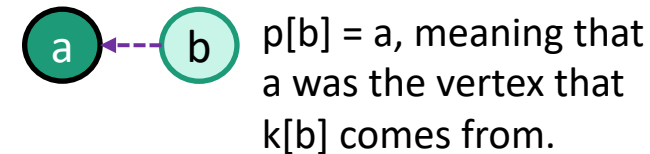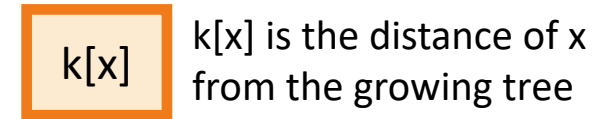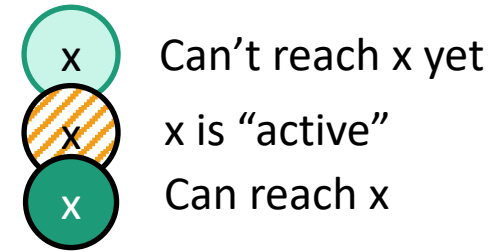

67

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 unreached 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.**

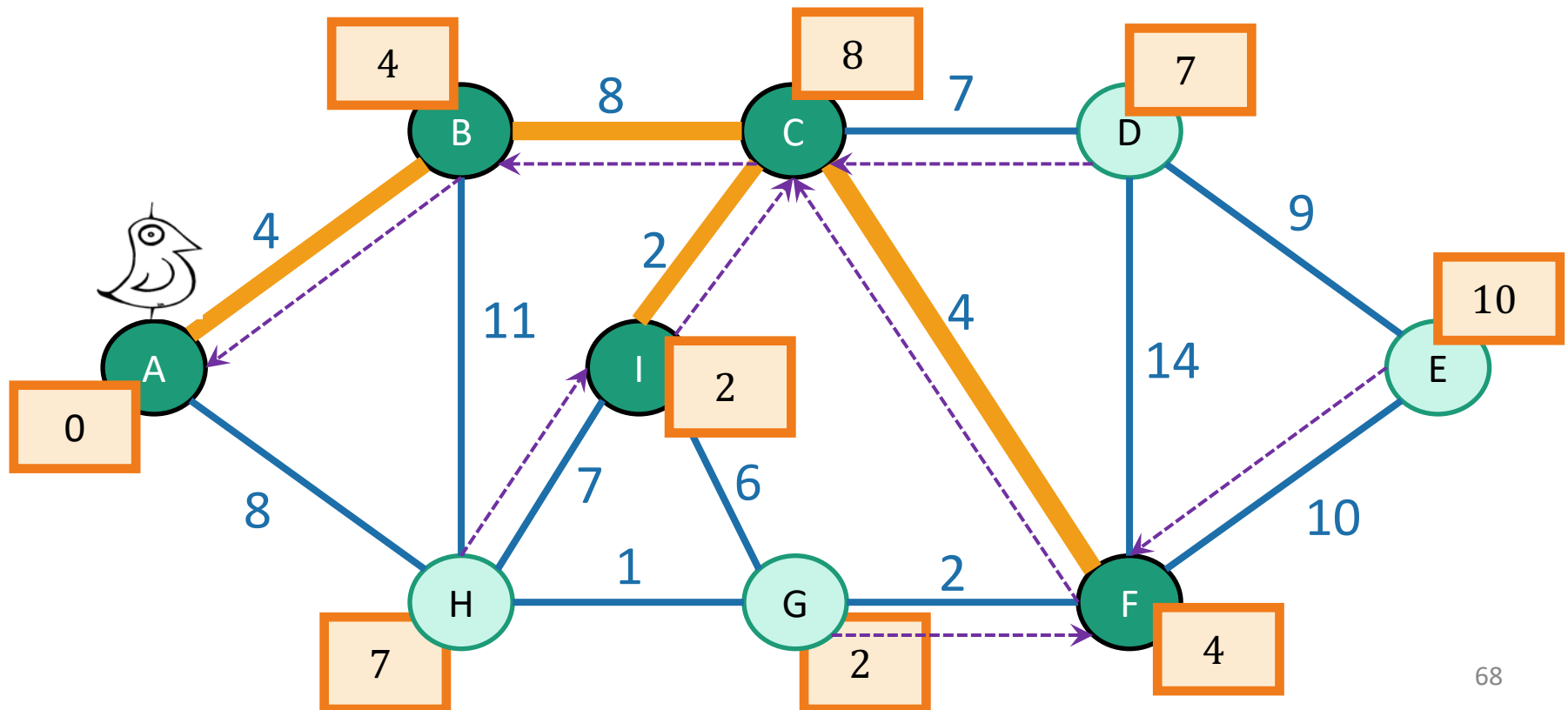x — Can't reach x yet

x — x is "active"

x — Can reach x

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

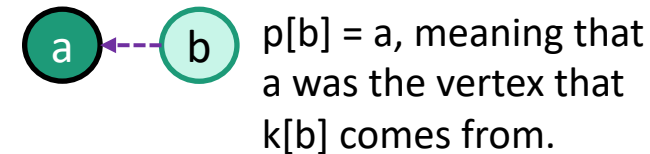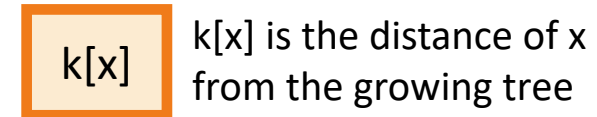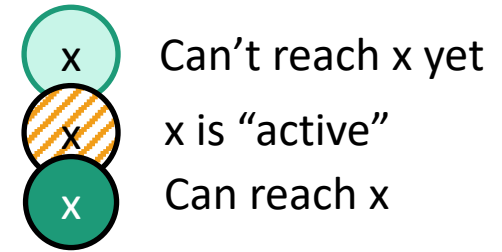a ← b — p[b] = a, meaning that a was the vertex that k[b] comes from.

Efficient implementation
Every vertex has a key and a parent

Until all the vertices are **reached:**

- Activate the **unreached** vertex u with the smallest key.
- **for each** of u's unreached 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.**

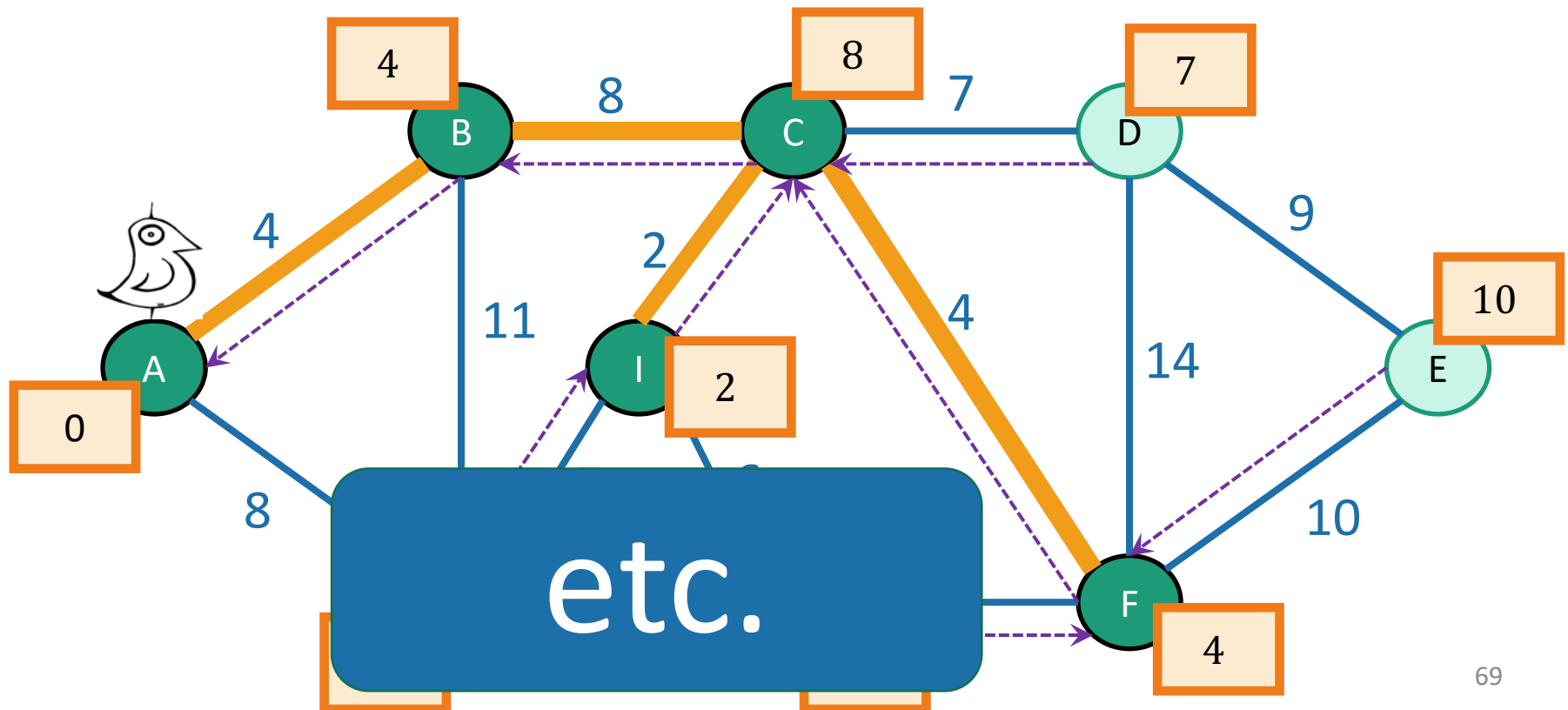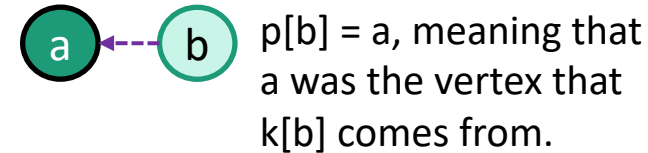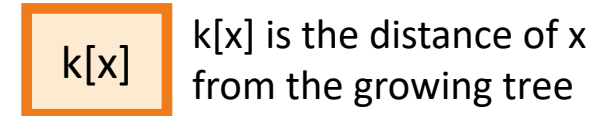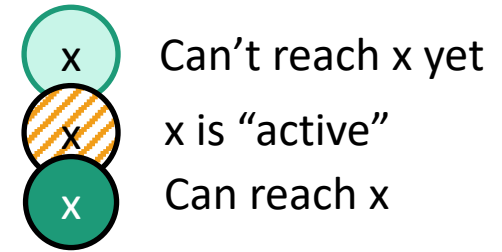Can't reach x yet

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.

etc.

69

Running time

Q : heap data structure

MST-PRIM(G, w, r)

1   **for** each $u \in G.V$
2       $u.key = \infty$
3       $u.\pi = \text{NIL}$
4   $r.key = 0$
5   $Q = G.V$
6   **while** $Q \neq \emptyset$
7       $u = \text{EXTRACT-MIN}(Q)$
8   **for** each $v \in G.Adj[u]$
9   **if** $v \in Q$ and $w(u, v) < v.key$
10              $v.\pi = u$
11              $v.key = w(u, v)$

Build-min-heap   : O(V)

|V| times

O(log V)

Total time for calling extract-min : O(v logv)
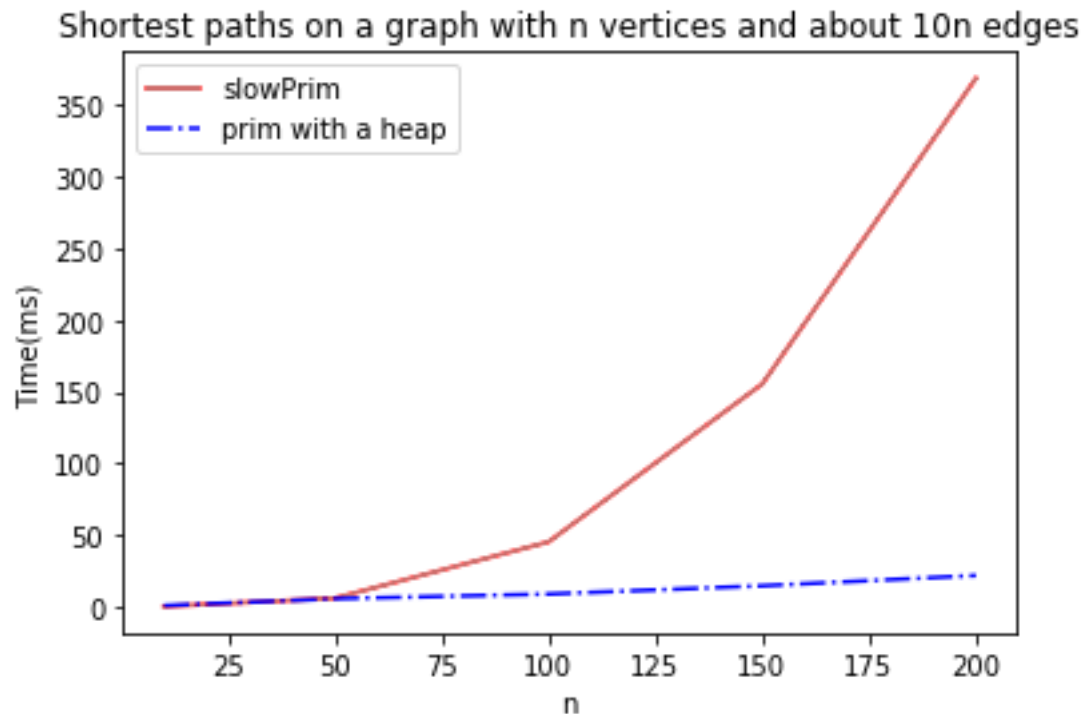
sum of the lengths of all adjacency lists : 2 |E|

O (E)

Constant time

Decrease-key : min heap : O(log V)

O (V logV + E logV) = O (E logV)

Running time

- Exactly the same as Dijkstra:
  - O (E logV) using a binary min-Heap as a priority queue.
  - O(E + V log(V)) amortized time if we use a Fibonacci Heap.

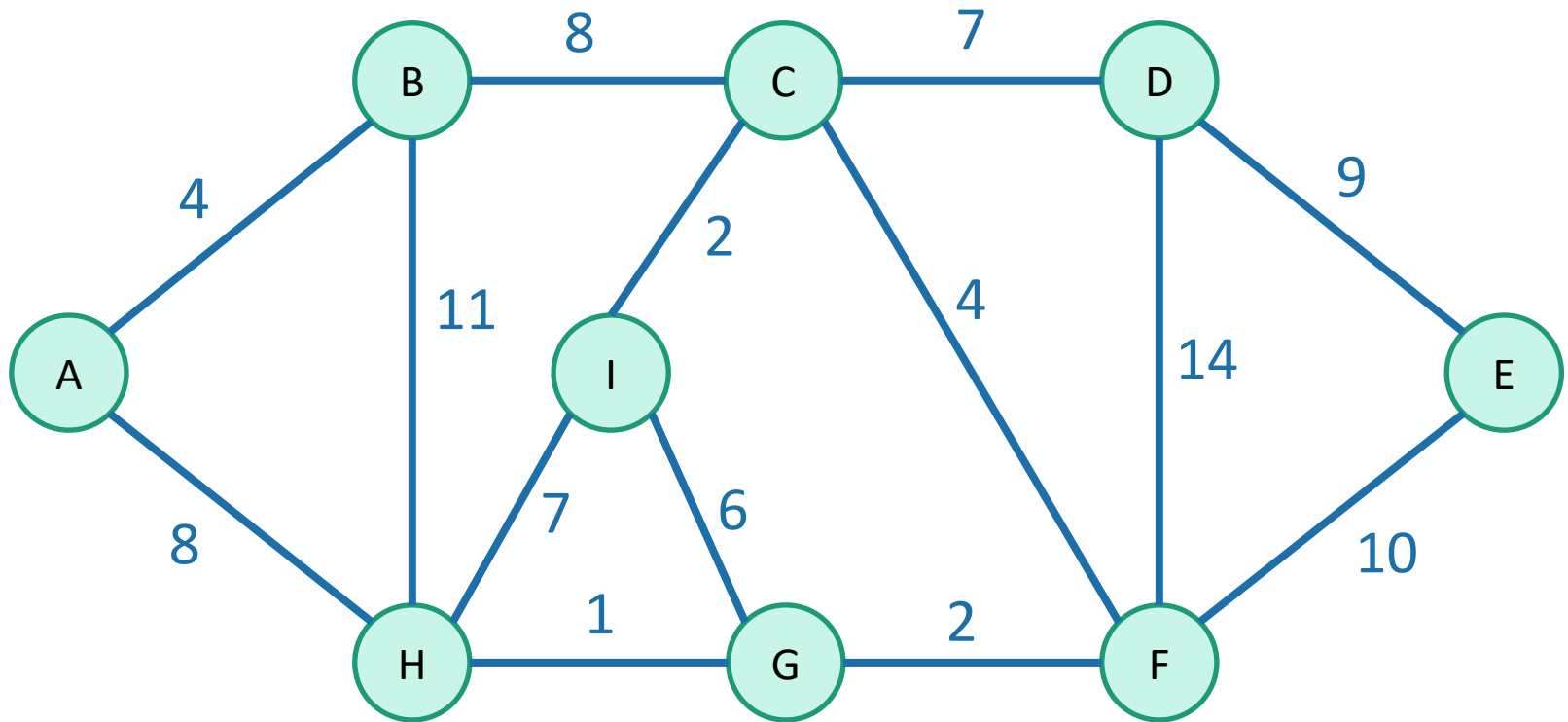Shortest paths on a graph with n vertices and about 10n edges

What have we learned?

- Prim's algorithm greedily grows a tree
  - smells a lot like Dijkstra's algorithm
- It finds a Minimum Spanning Tree!
  - in time **O(mlog(n))** if we implement it with a binary min-heap.
  - In amortized time **O(m + nlog(n))** with a Fibonacci heap.
- 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.**

That's not the only greedy algorithm for MST!

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

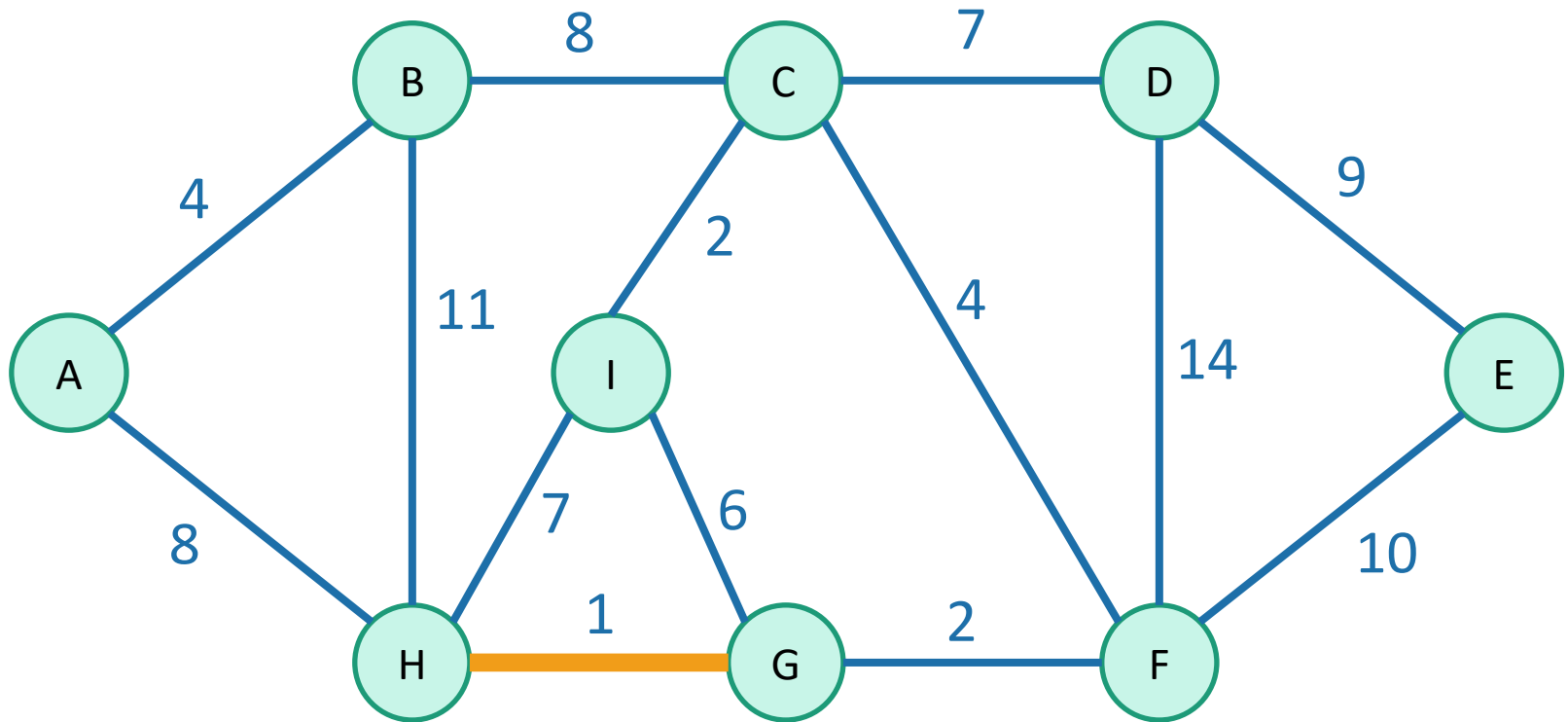what if we just always take the cheapest edge?
whether or not it's connected to what we have so far?

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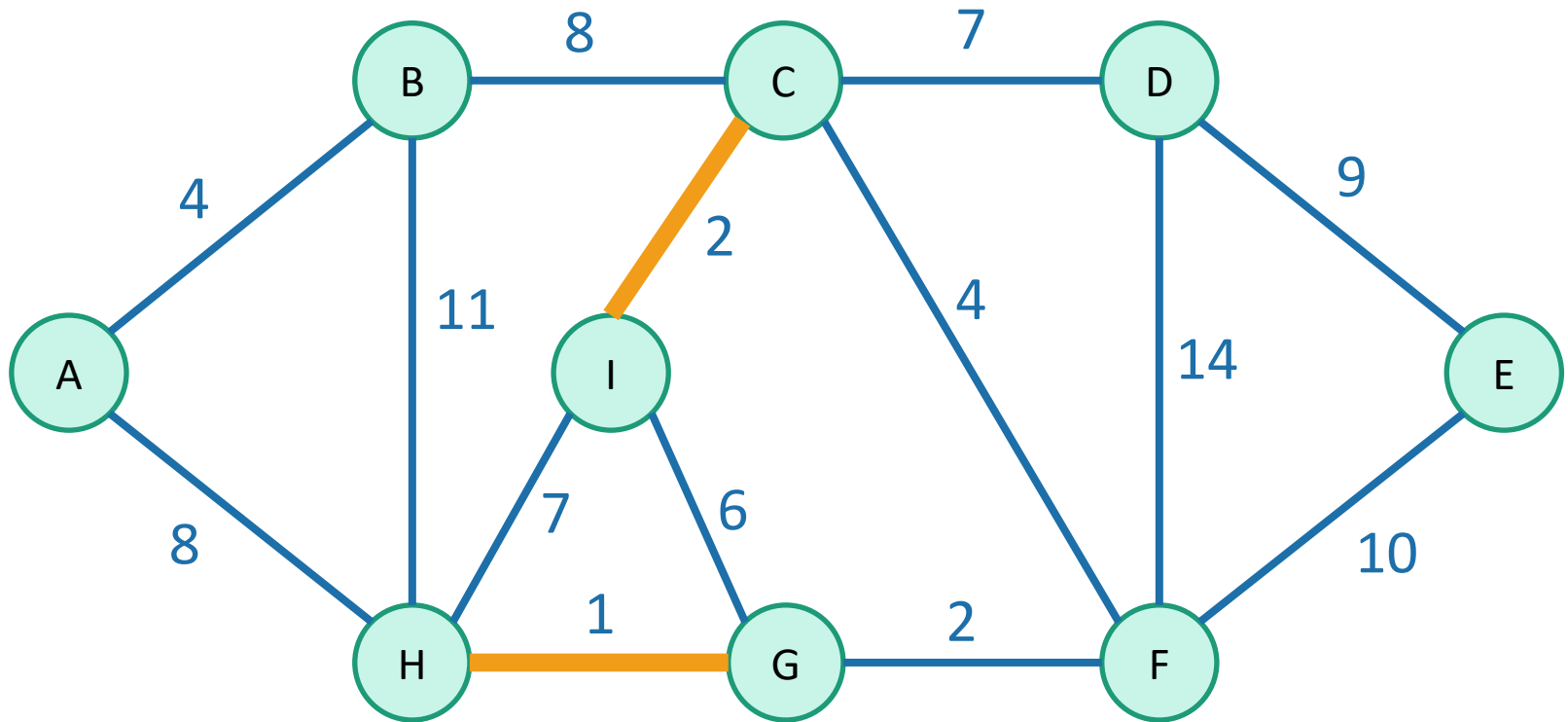what if we just always take the cheapest edge?
whether or not it's connected to what we have so far?

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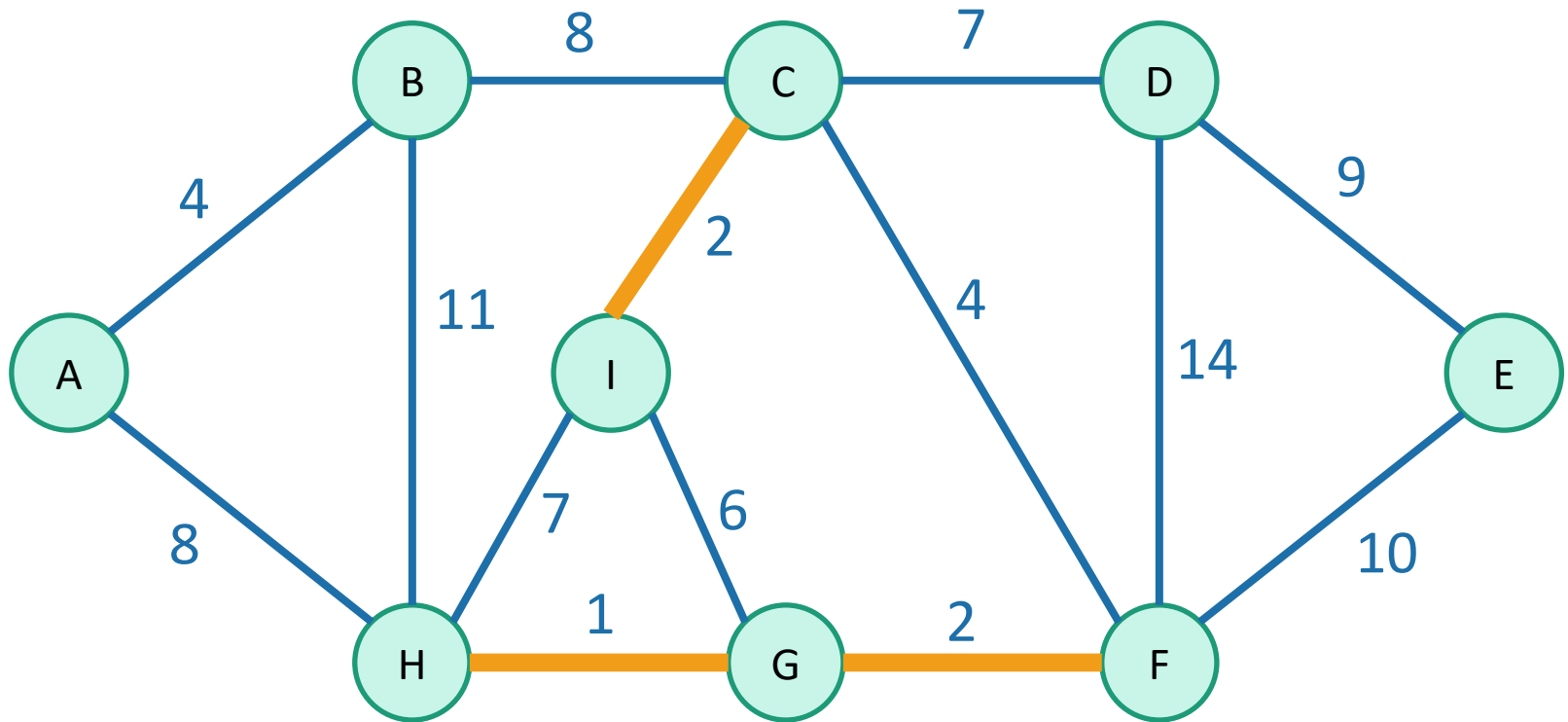what if we just always take the cheapest edge?
whether or not it's connected to what we have so far?

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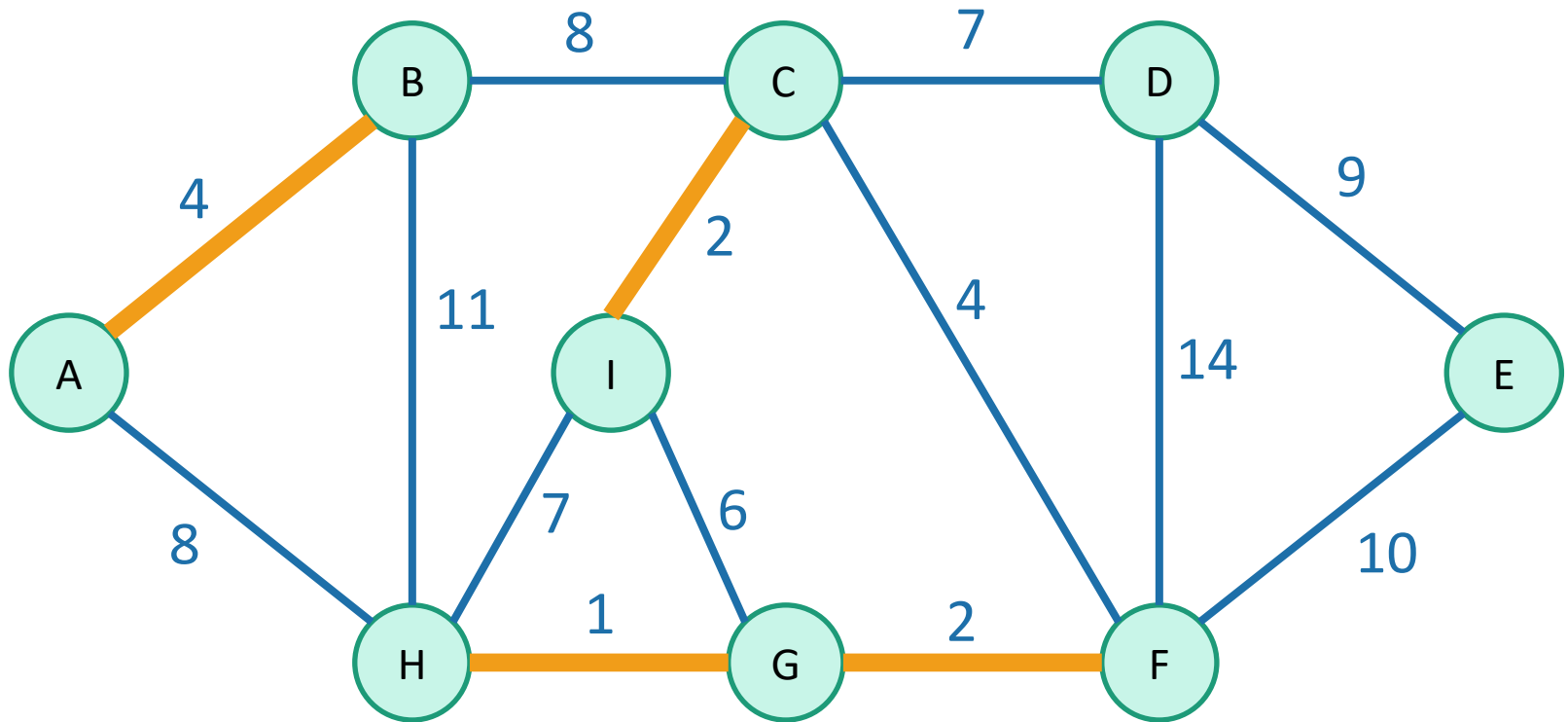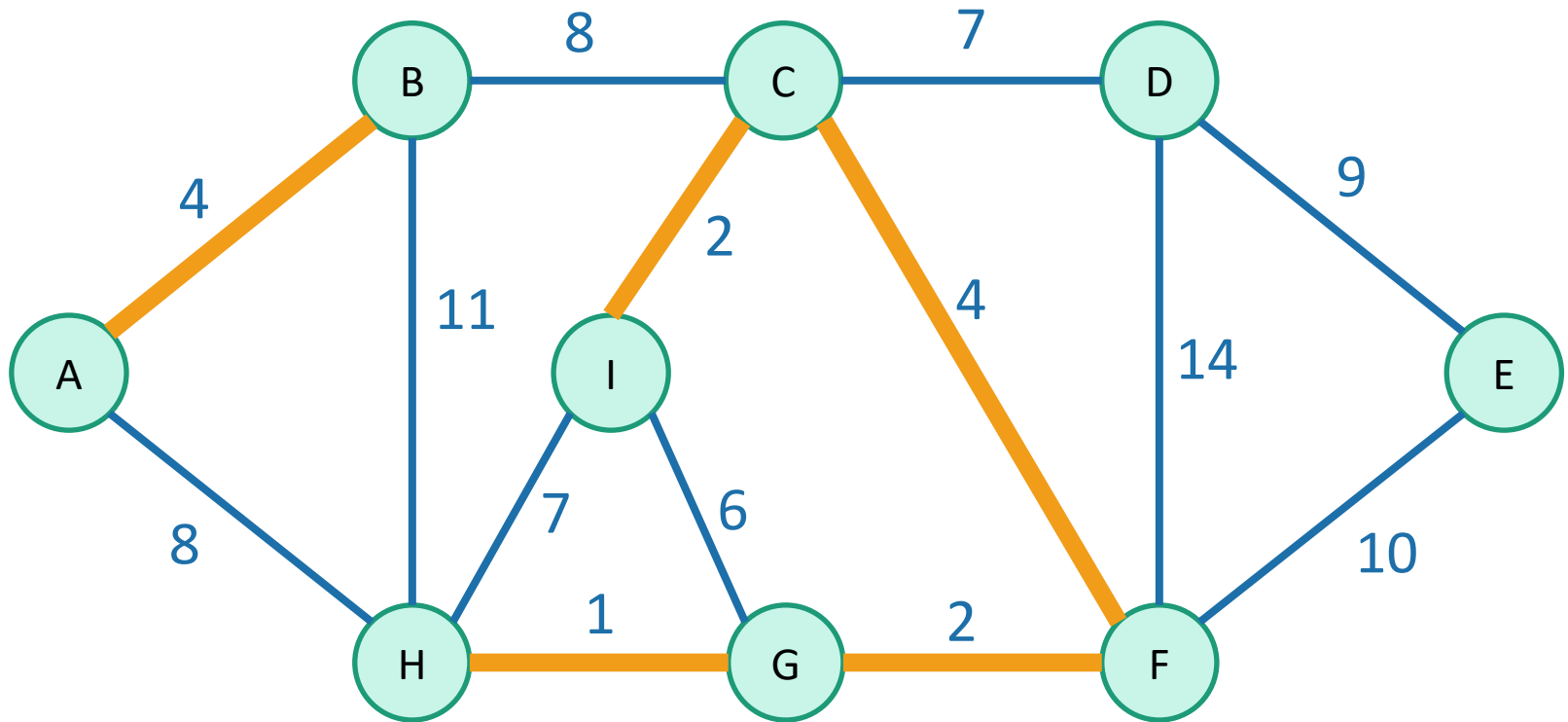what if we just always take the cheapest edge?
whether or not it's connected to what we have so far?

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

8

7

B — C — D

4

9

11

2

4

A

I

14

E

7

6

!!!!!

8

1

2

10

H

G

F

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

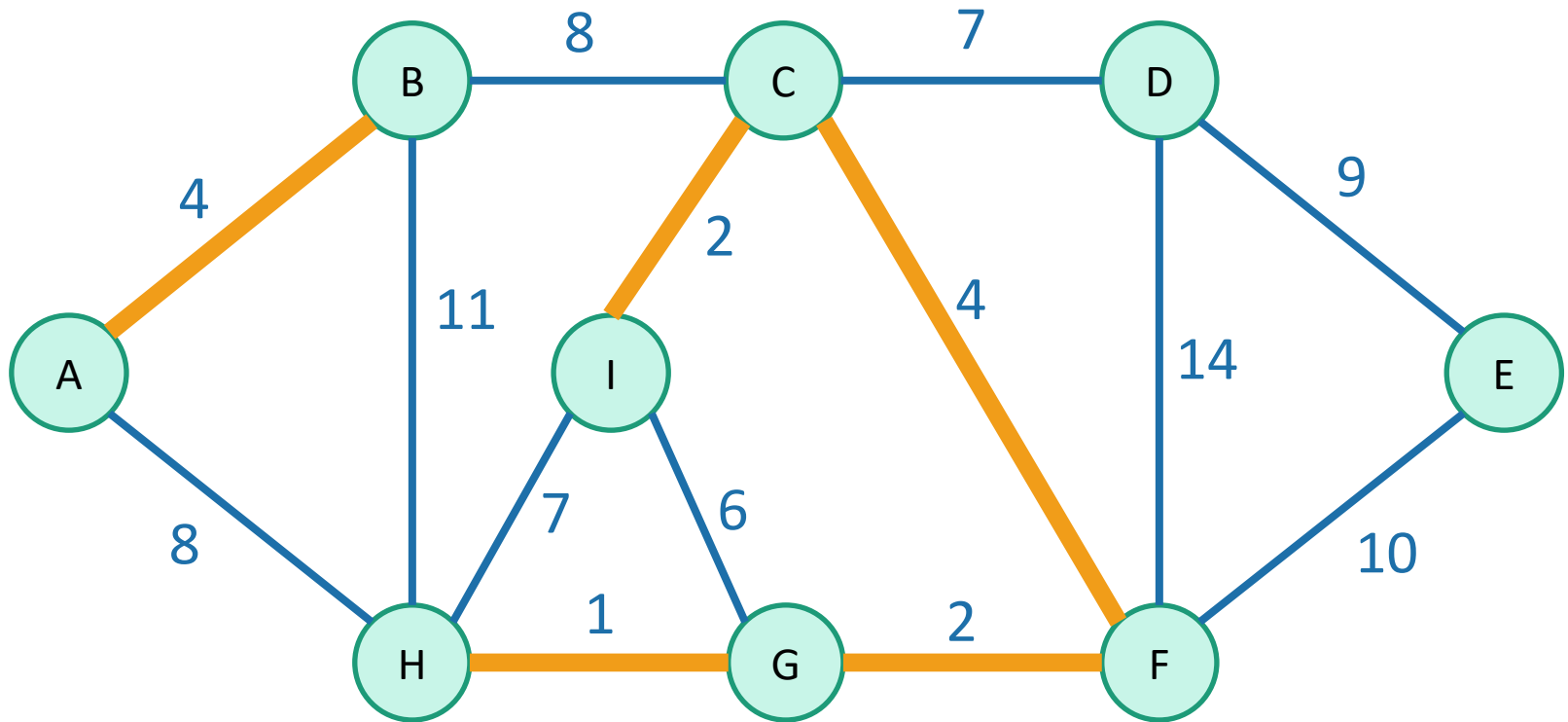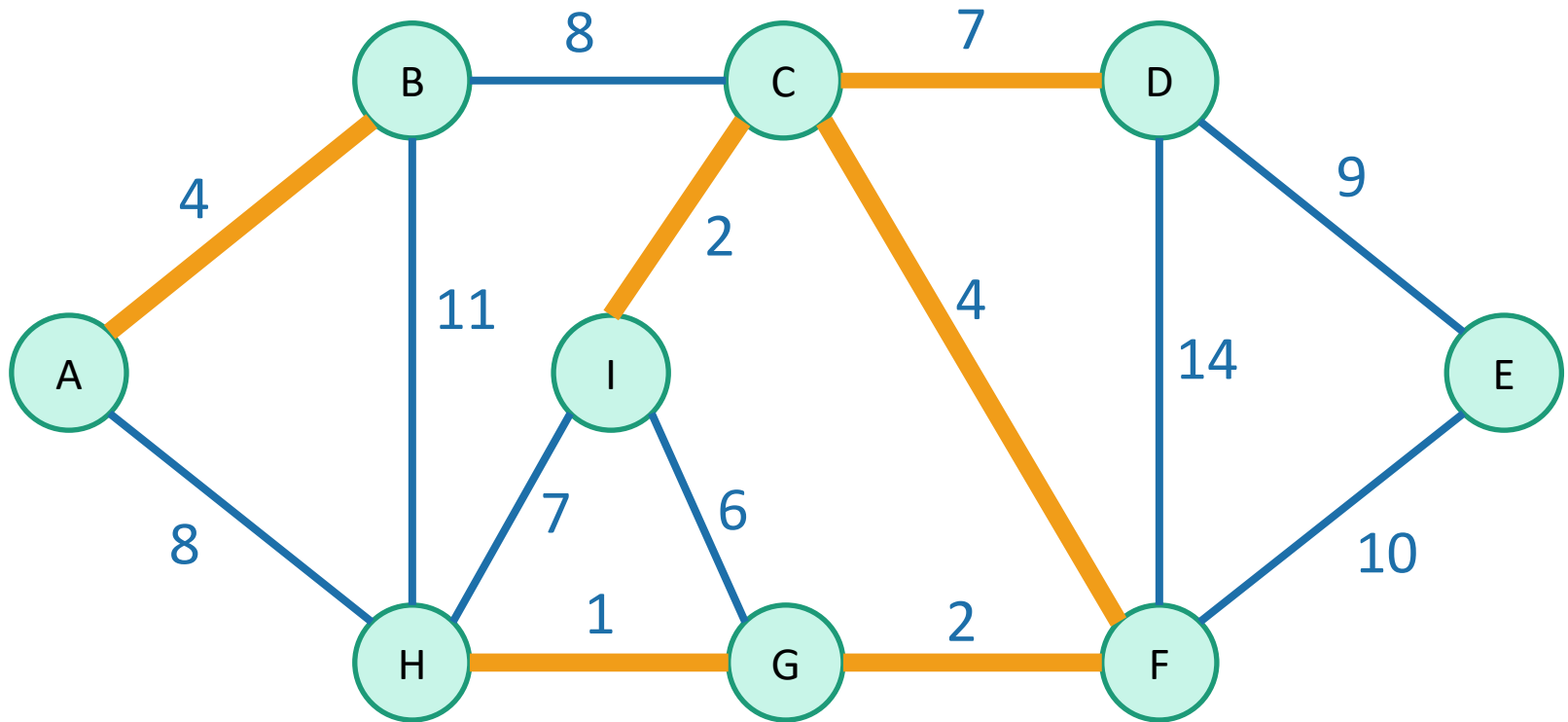# That's not the only greedy algorithm

what if we just always take the cheapest edge?
whether or not it's connected to what we have so far?
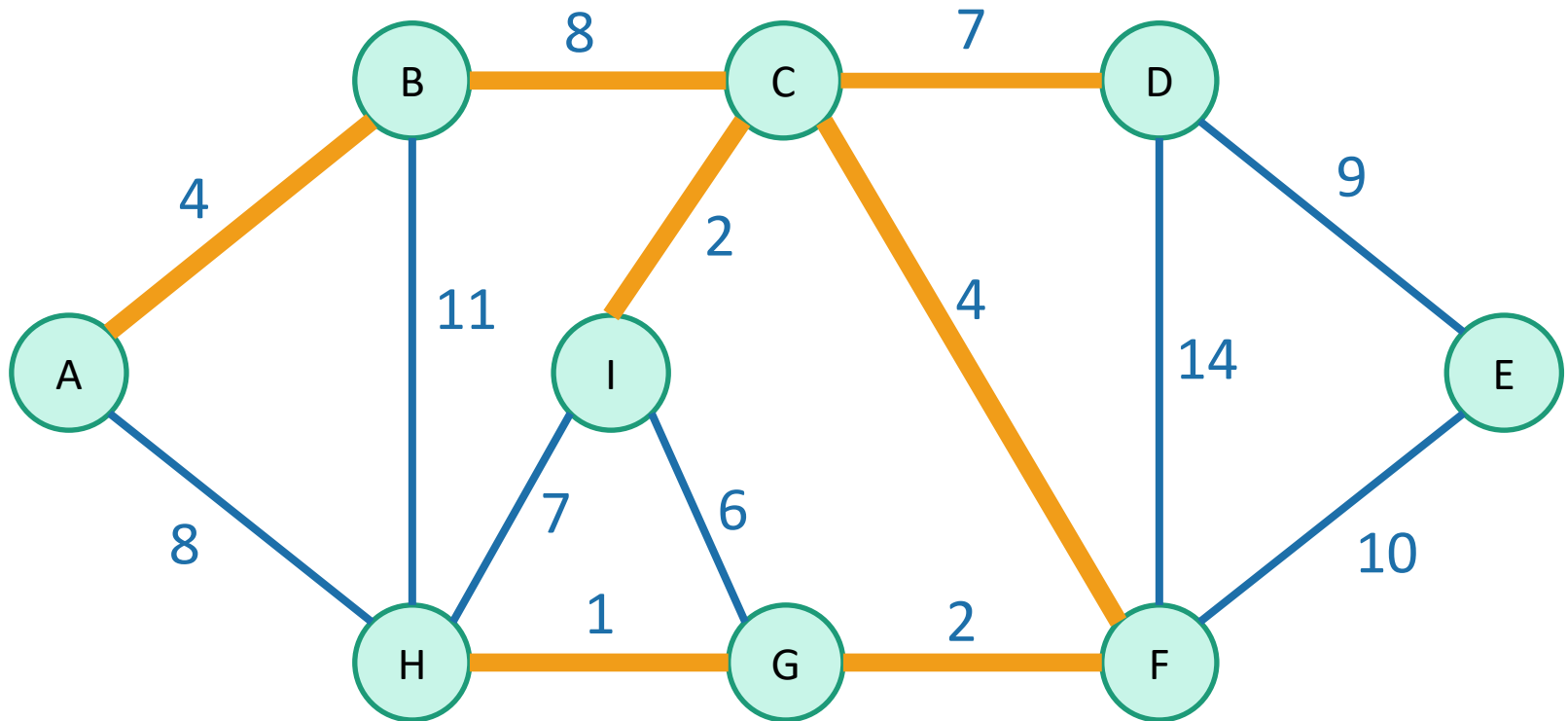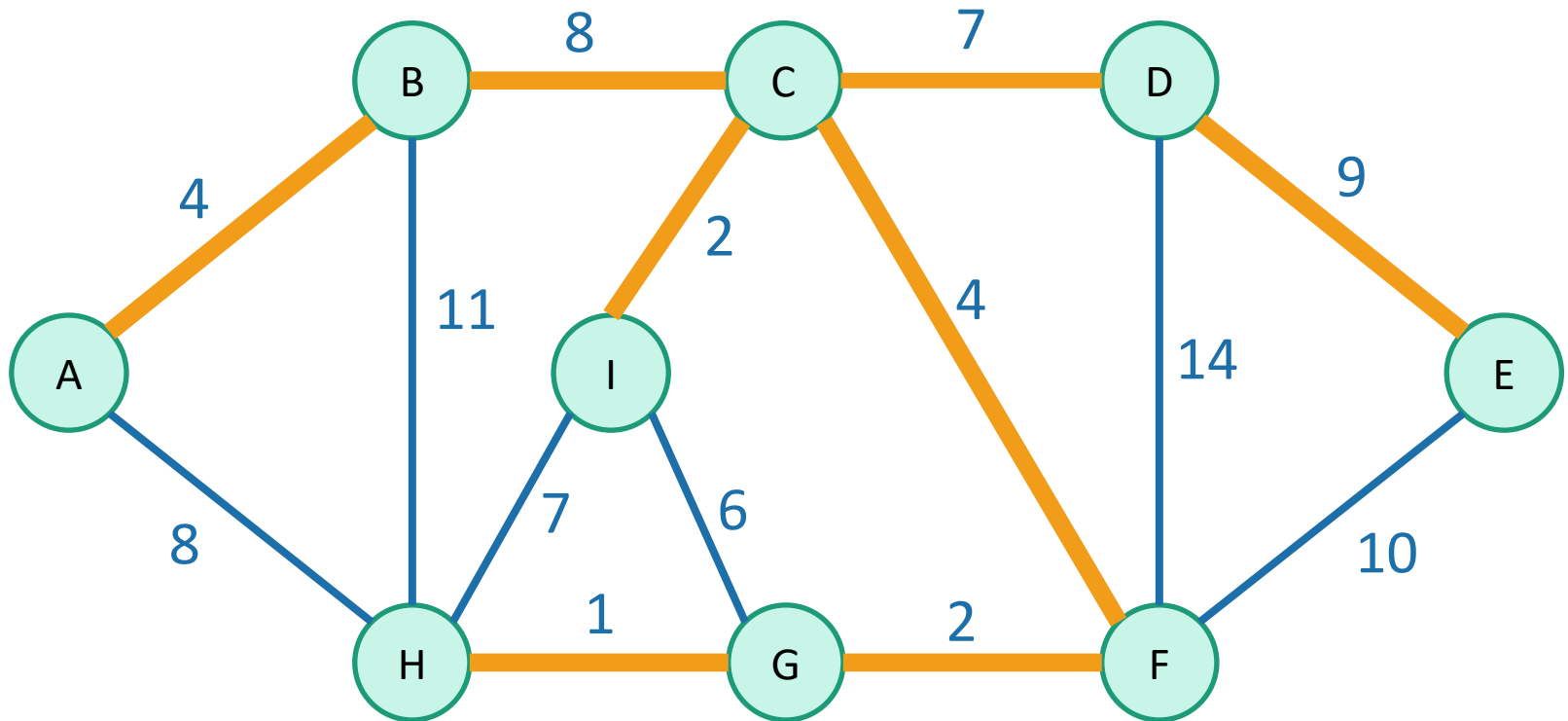
That won't
cause a cycle

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): ⟵ *m iterations through this loop*
        - **if** adding e to MST won't cause a cycle: ⟵
            - add e to MST. *How do we check this?*
- **return** MST

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

Naively, the running time is ???:
- 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?
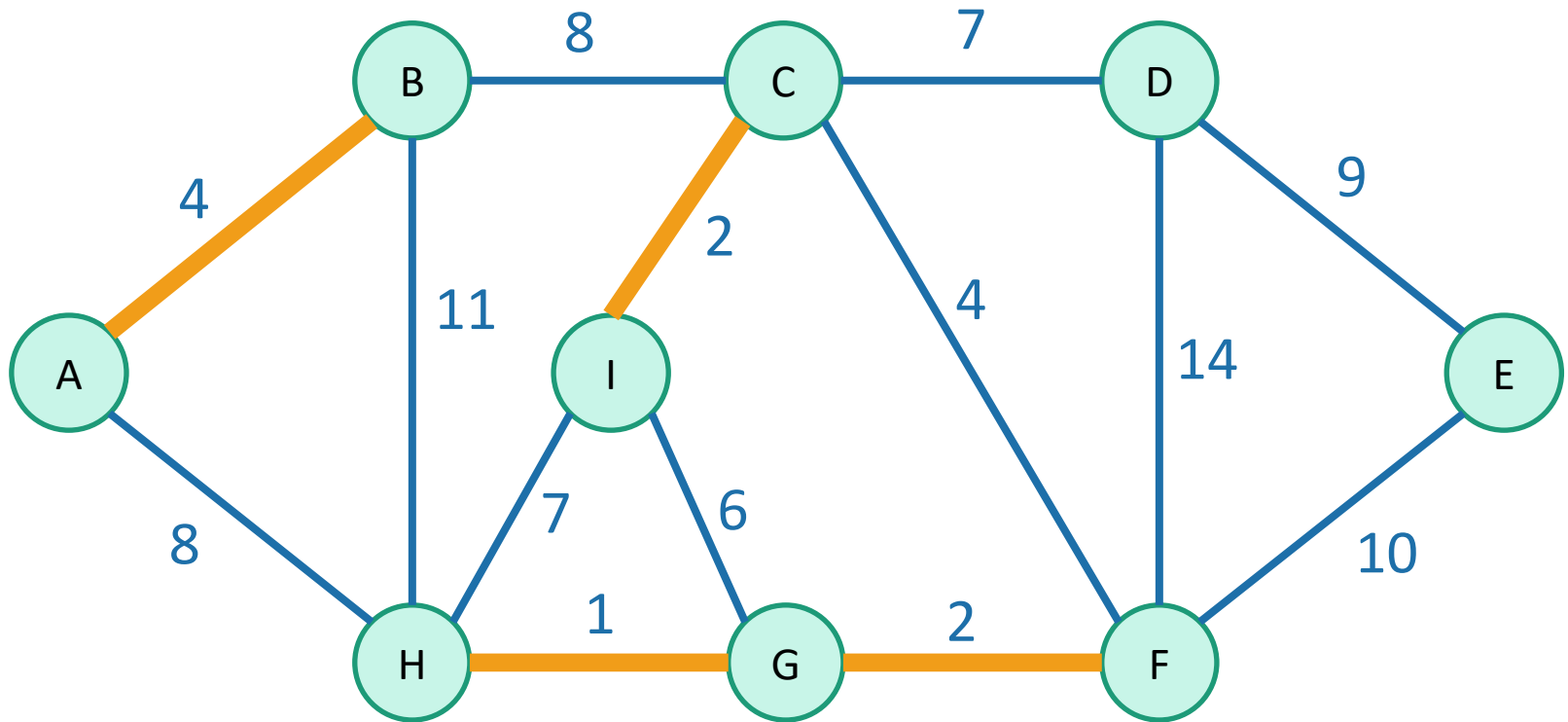   - the pseudocode above says "slowKruskal"...

Let's do this one first

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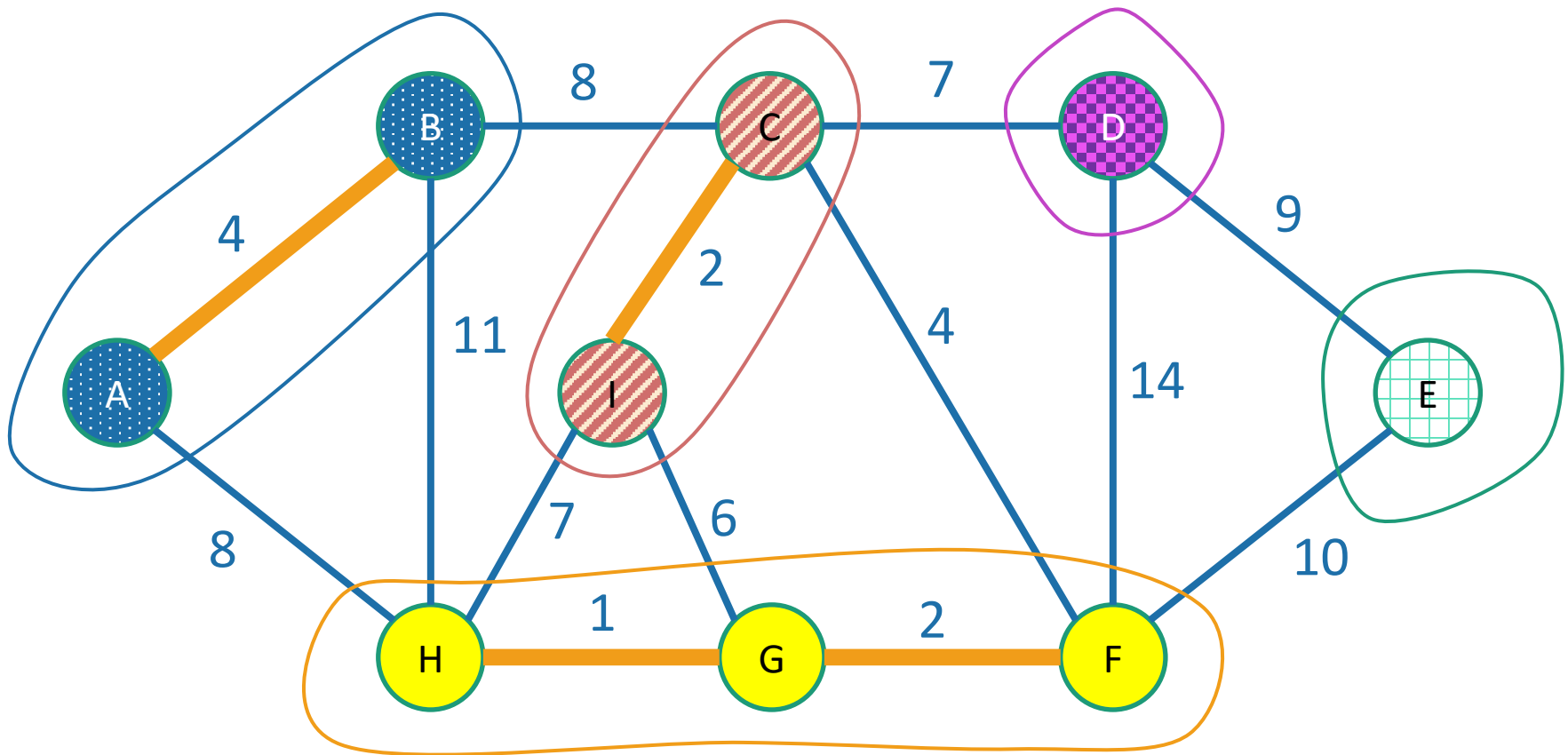
At each step of Kruskal's, we are maintaining a forest.

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

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

When we add an edge, we merge two trees:

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

When we add an edge, we merge two trees:

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

When we add an edge, we merge two trees:
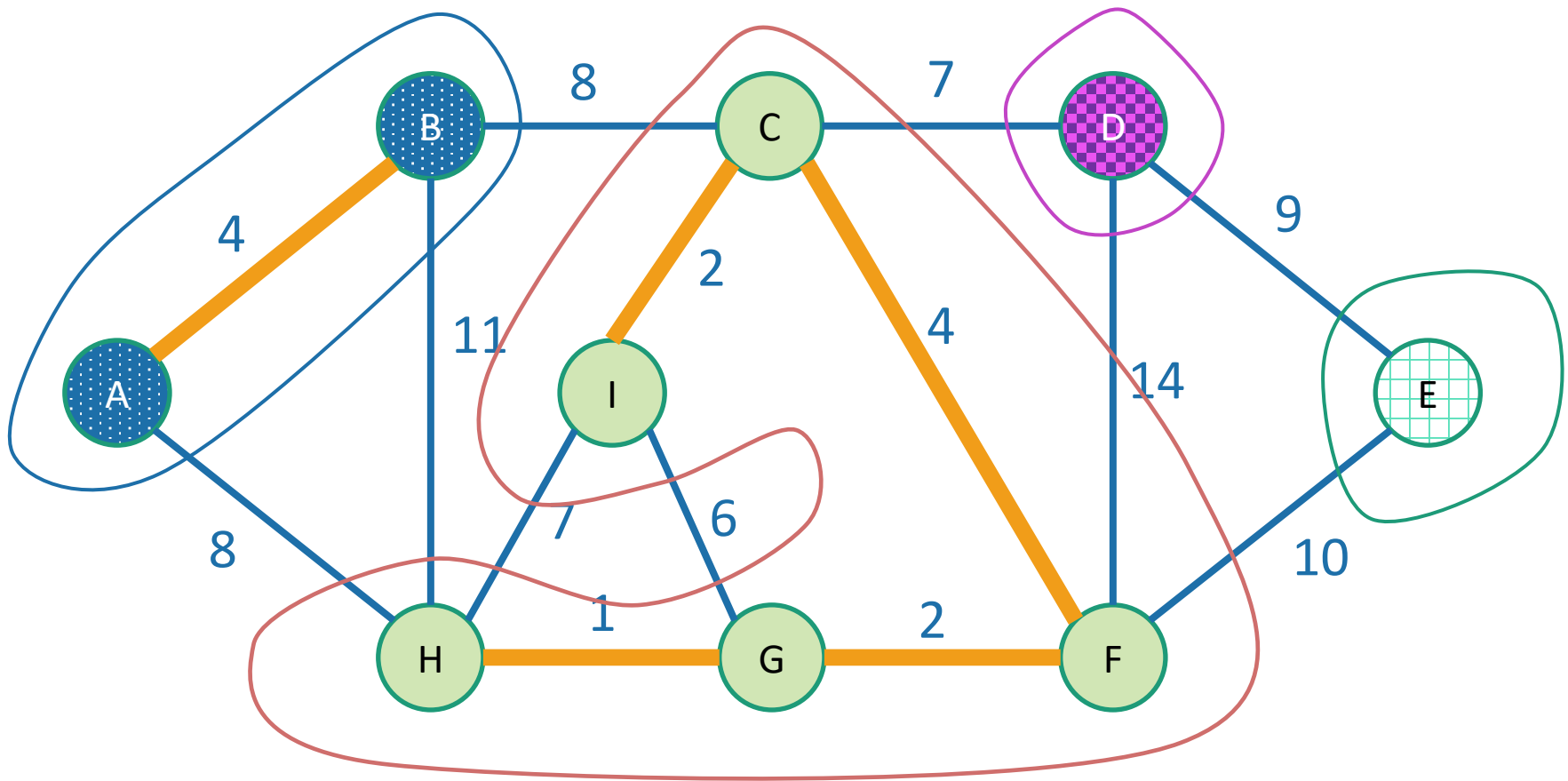
We never add an edge within a tree since that would create a cycle.

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)
```
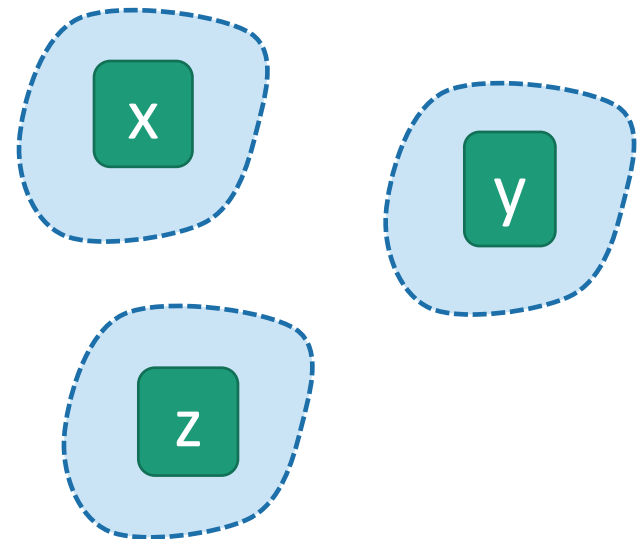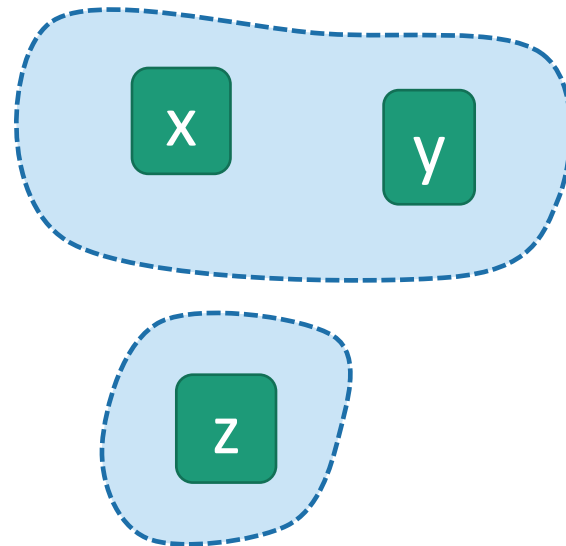
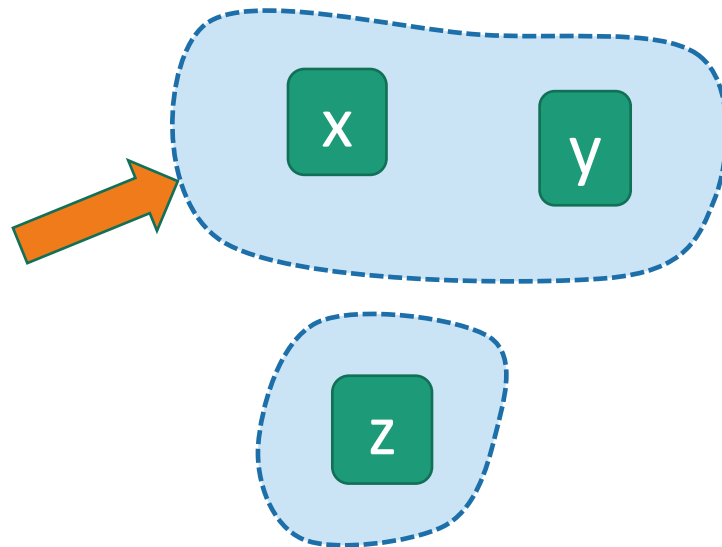# Union-find data structure
also called disjoint-set data structure

- 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.
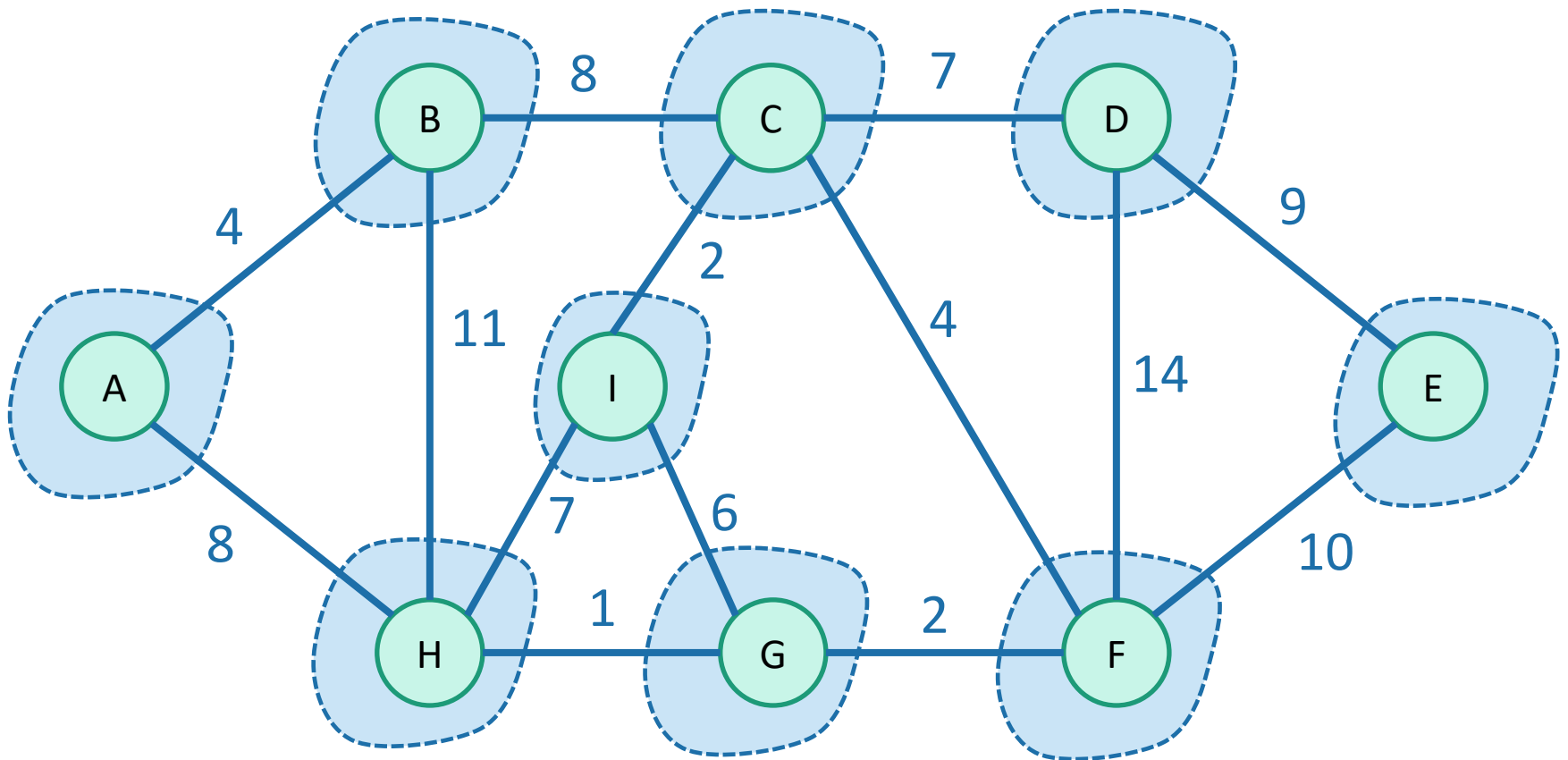
Once more…

Then start merging.

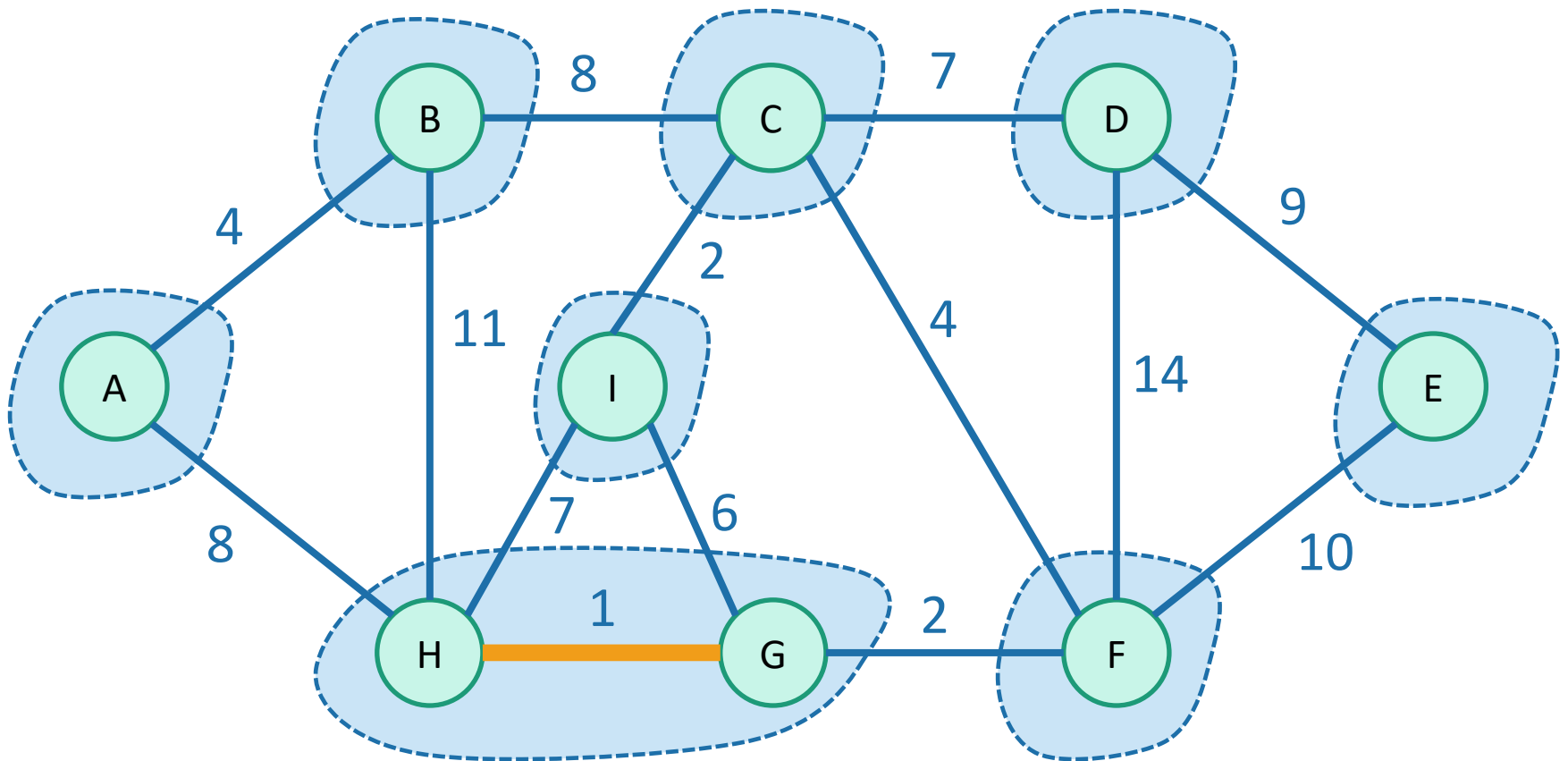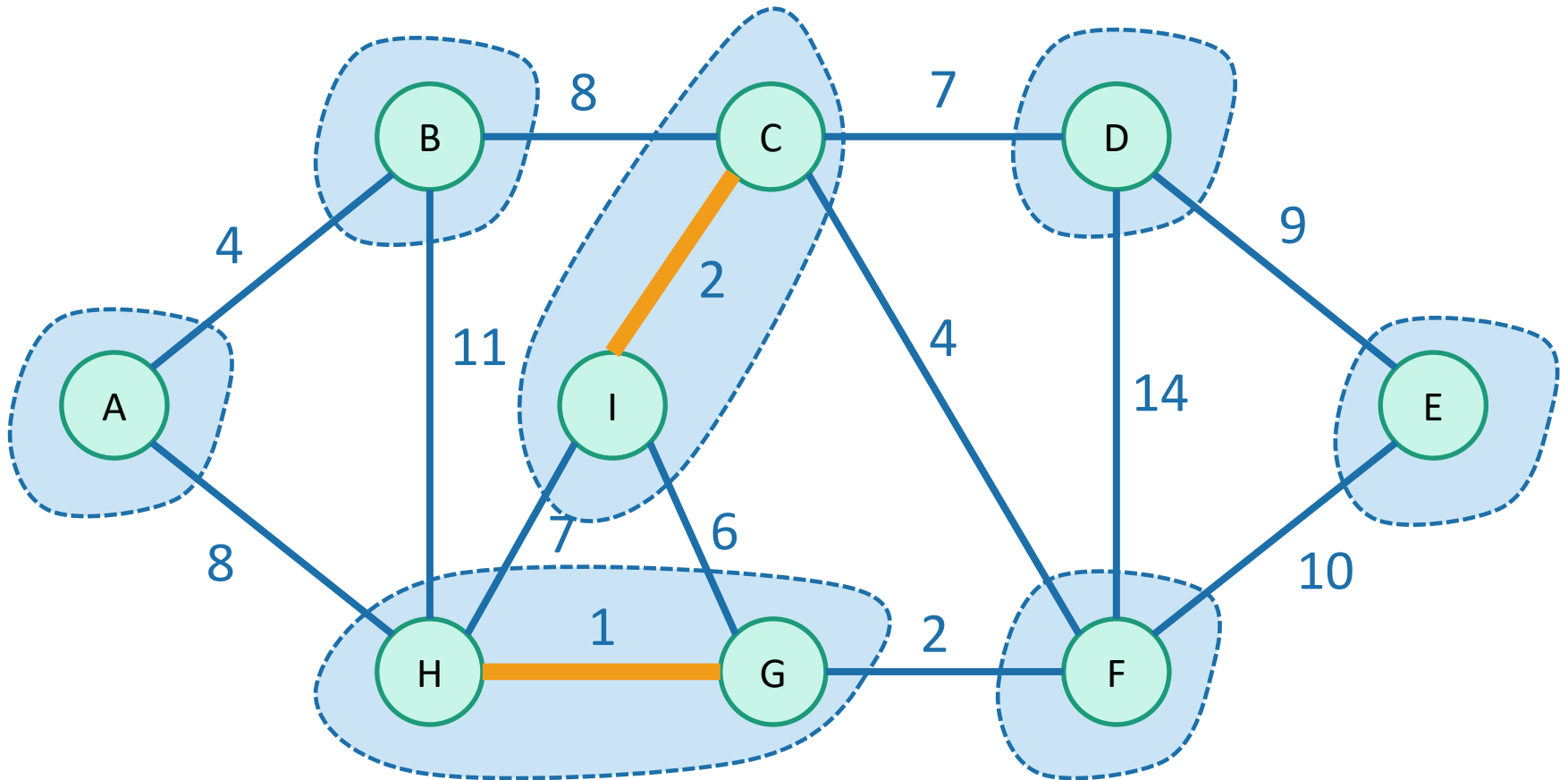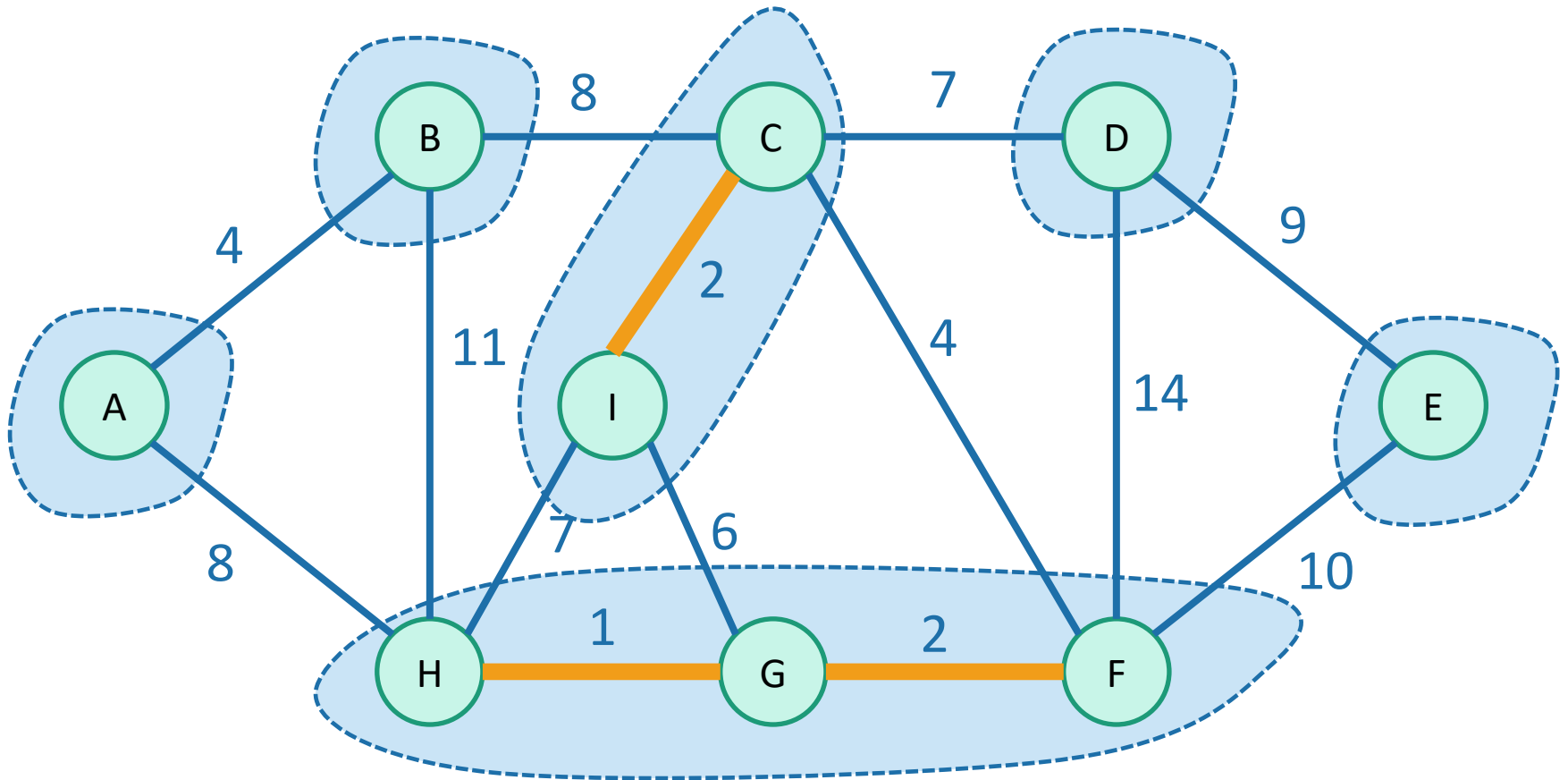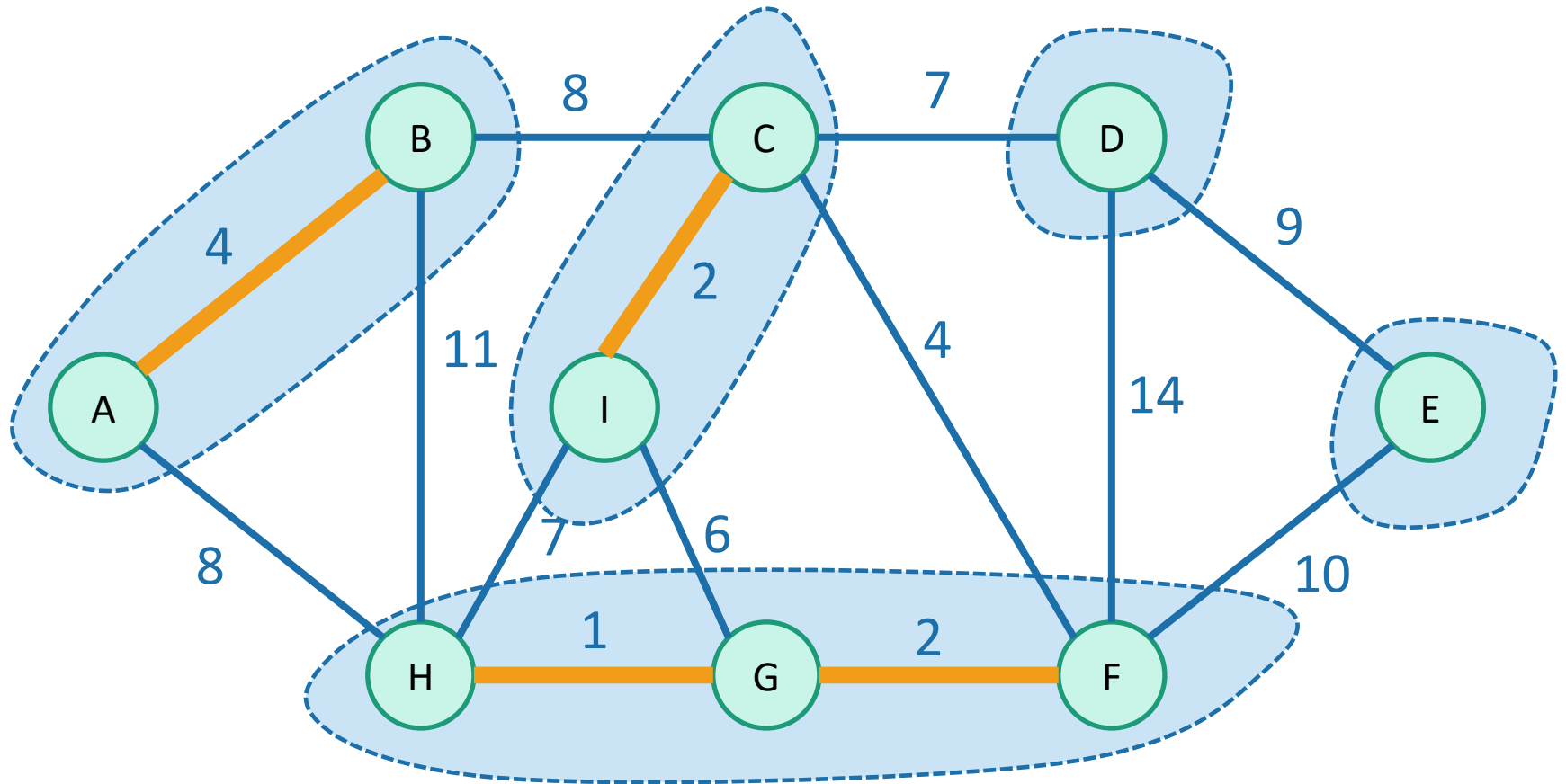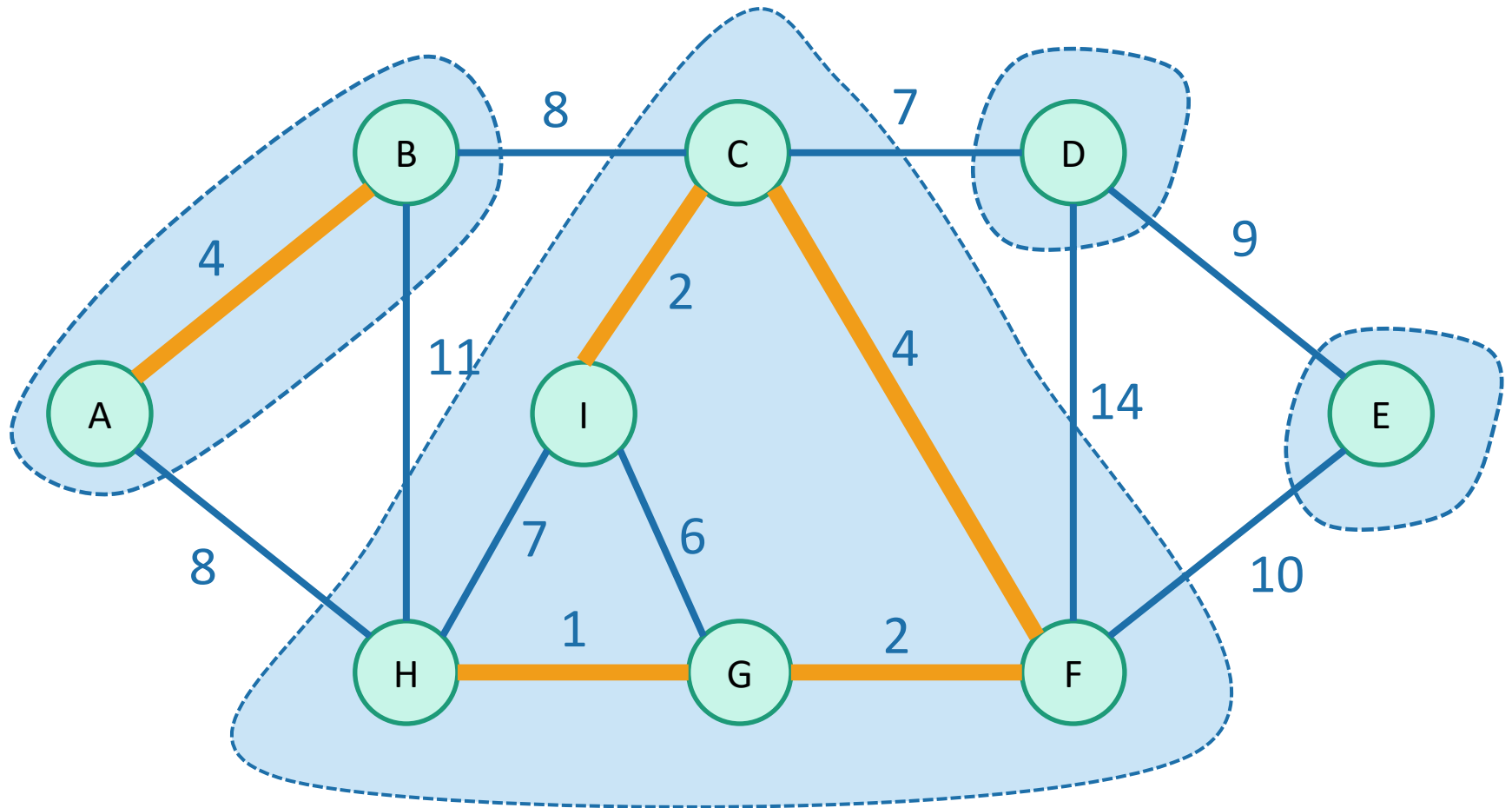# Once more…

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.

Once more…

Then start merging.

104
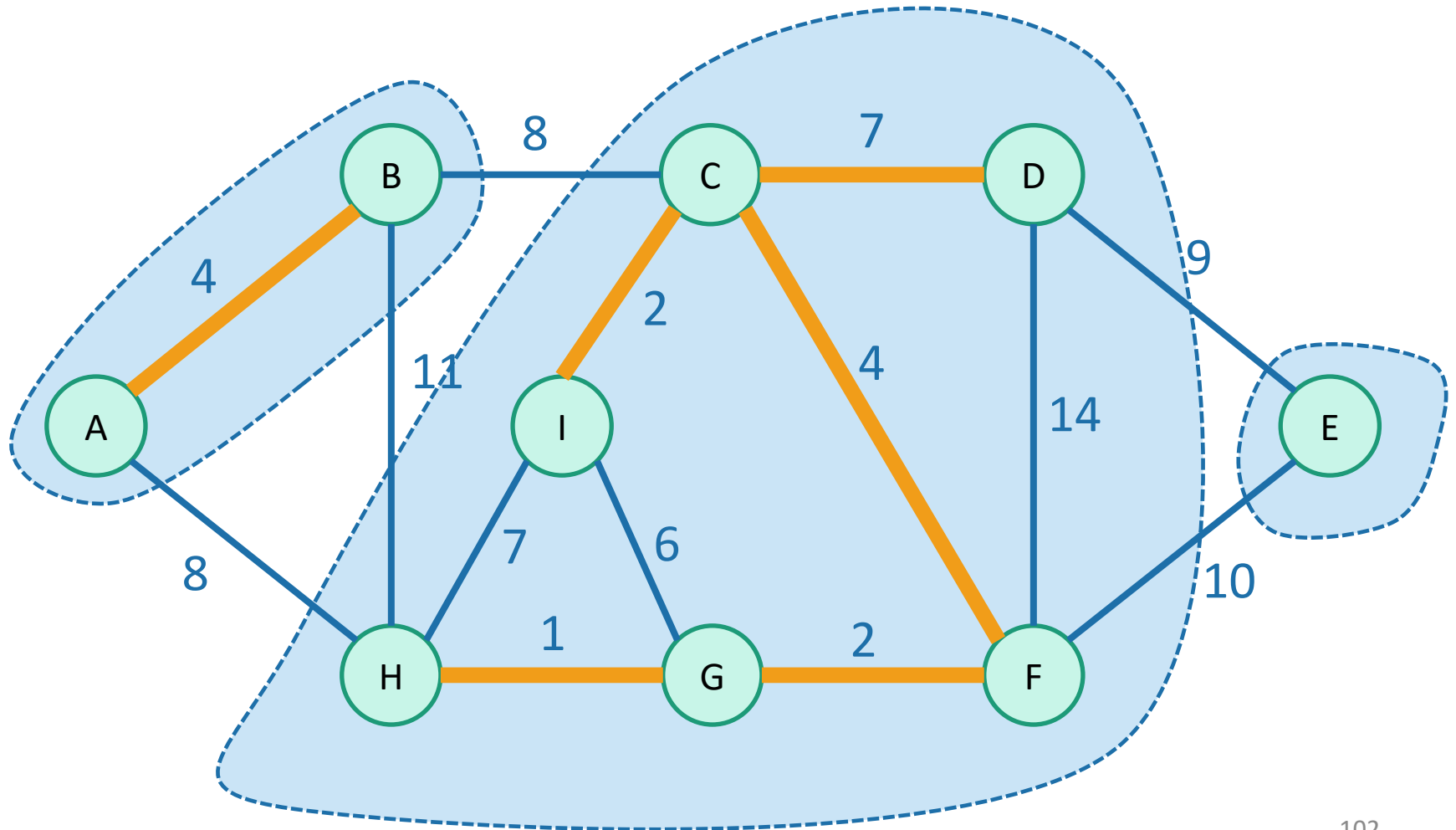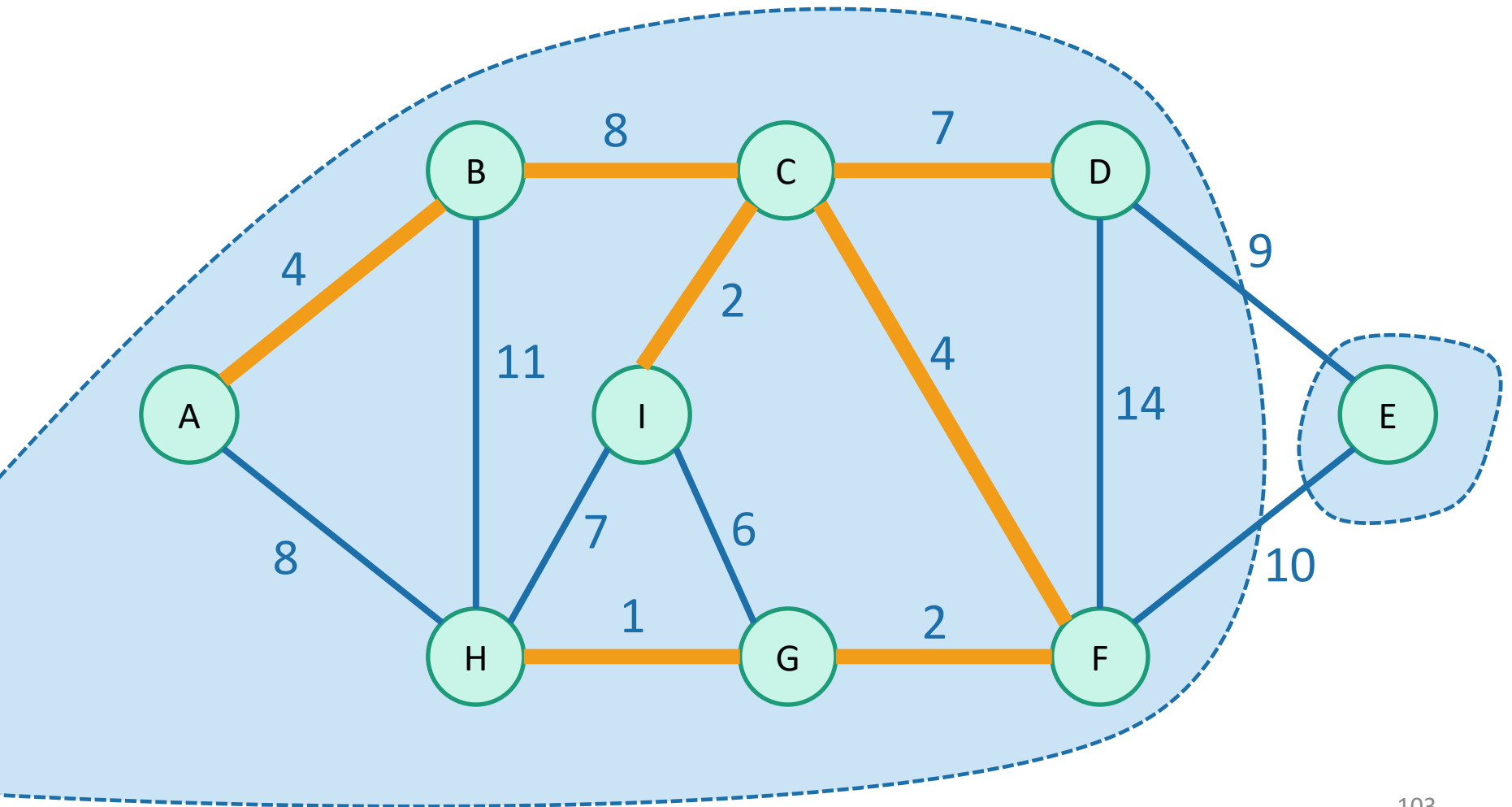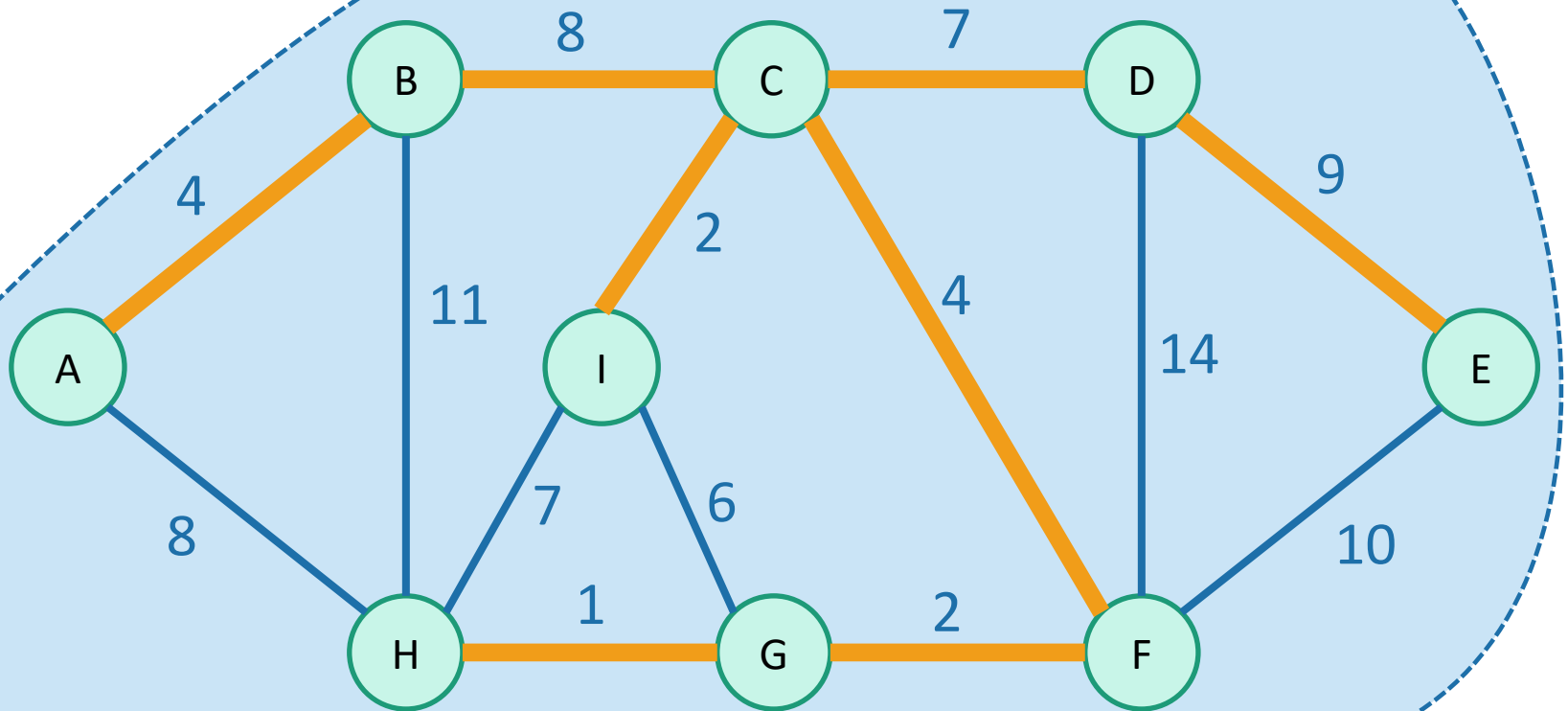
Running time

- Sorting the edges takes O(E log(E))
  - In practice, if the weights are small integers we can use radixSort and take time O(E)

- For the rest:
  - V calls to **makeSet**
    - put each vertex in its own set
  - 2E calls to **find**
    - for each edge, **find** its endpoints
  - V-1 calls to **union**
    - we will never add more than V-1 edges to the tree,
    - so we will never call **union** more than V-1 times.

In practice, each of makeSet, find, and union run in constant time*
(There is a simpler way which does find and union in time O(log n)).

- Total running time:
  - Worst-case O(E log(V)), just like Prim
  - Closer to O(E) if you can do radixSort

*technically, they run in *amortized time* $O(\alpha(n))$, where $\alpha(n)$ is the *inverse Ackerman function*. $\alpha(n) \leq 4$ provided that n is smaller than the number of atoms in the universe.

Two questions

1. Does it work?
   - That is, does it actually return a MST?
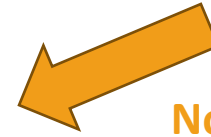
   Now that we understand this "tree-merging" view, let's do this one.

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}}

This edge is light

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

108

Partway through Kruskal

- Assume that our choices **S** so far don't rule out success.
  - There is an MST extending them
- The **next edge** we add will merge two trees, **T1, T2**

S is the set of edges selected so far.

109
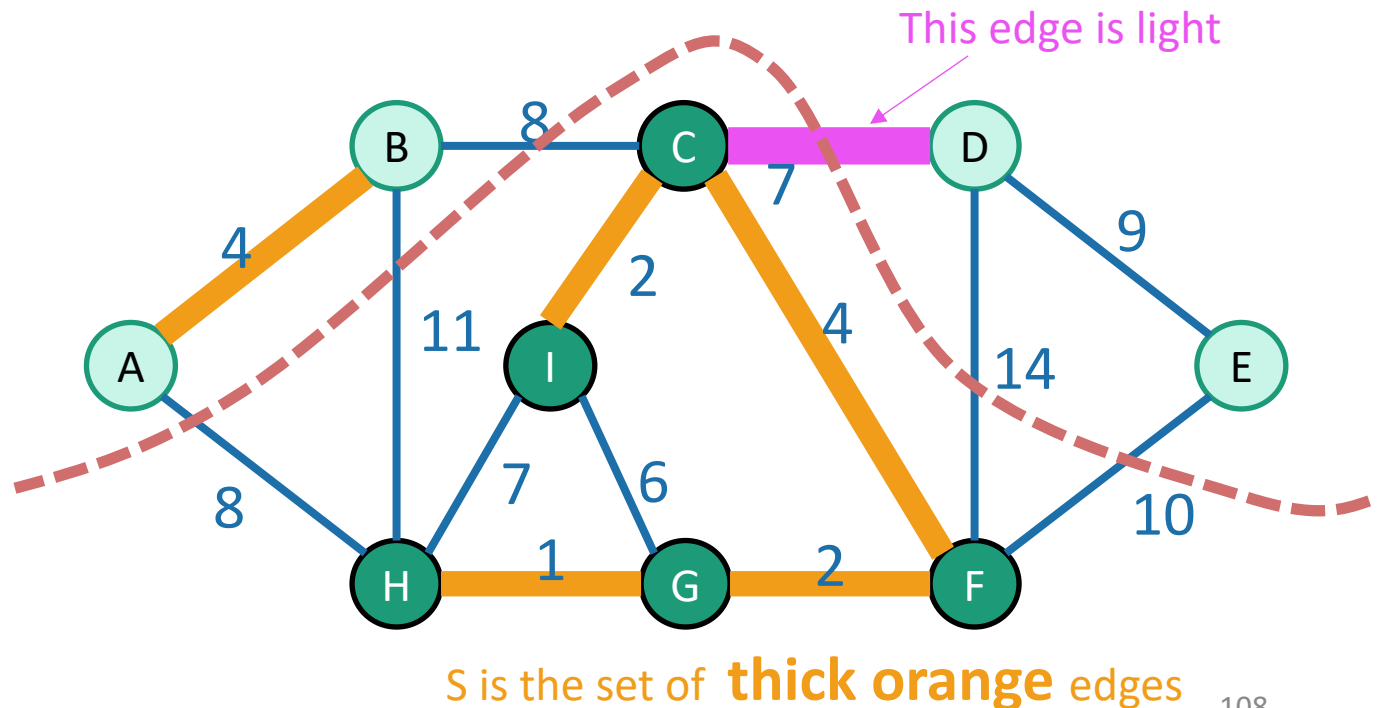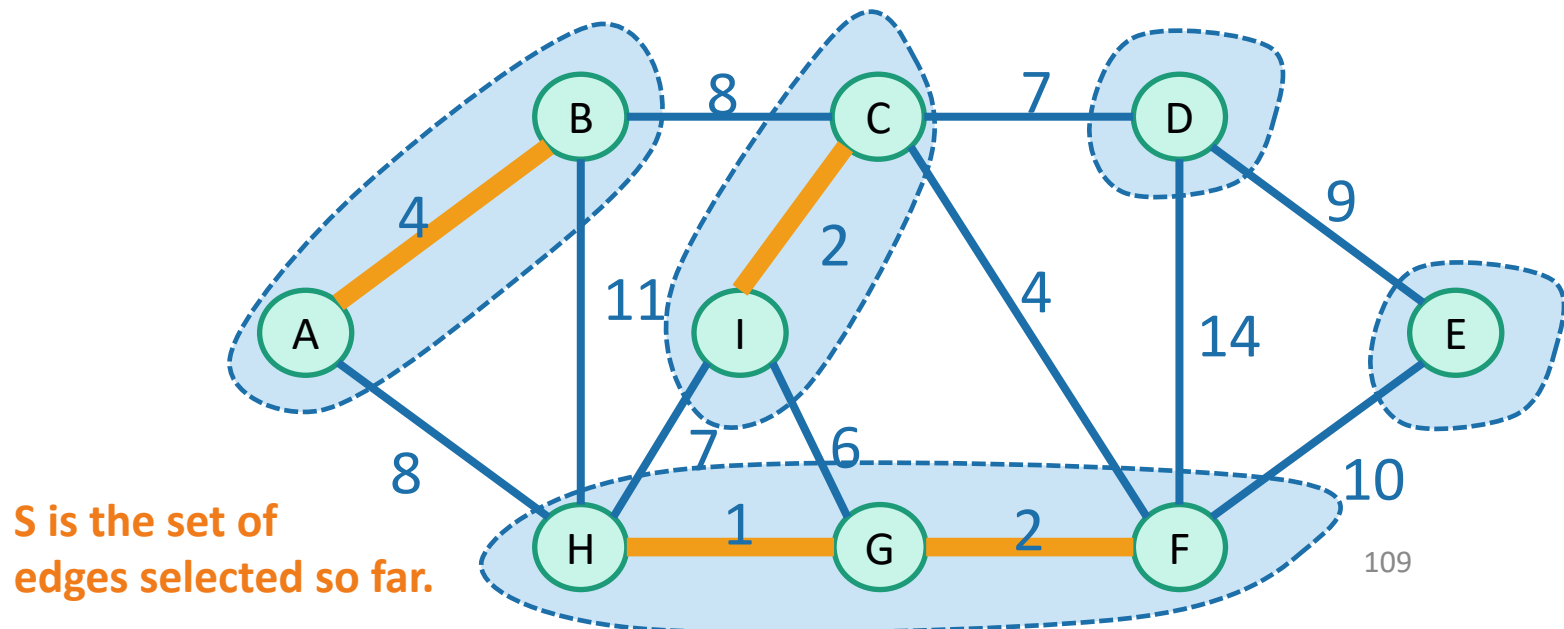
Partway through Kruskal

- Assume that our choices **S** so far don't rule out success.
  - There is an MST extending them
- The **next edge** we add will merge two trees, **T1, T2**

How can we use our lemma to show that our next choice also does not rule out success?

This is the next edge

S is the set of edges selected so far.

Partway through Kruskal

- Assume that our choices **S** so far don't rule out success.
  - There is an MST extending them
- The **next edge** we add will merge two trees, **T1, T2**
- Consider the cut **{T1, V − T1}.**
  - This cut respects S
  - Our **new edge is light** for the cut

This is the next edge

S is the set of edges selected so far.

Partway through Kruskal

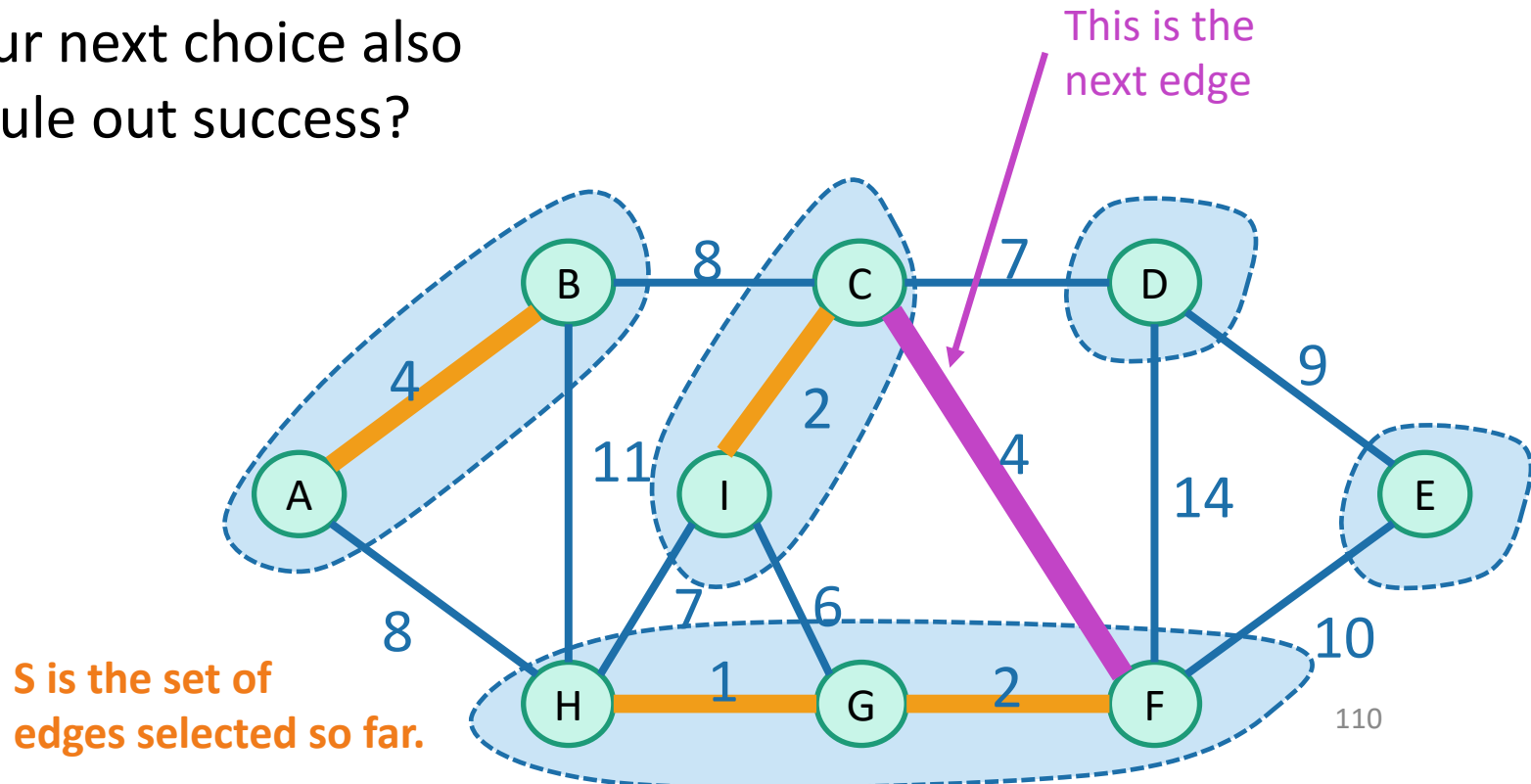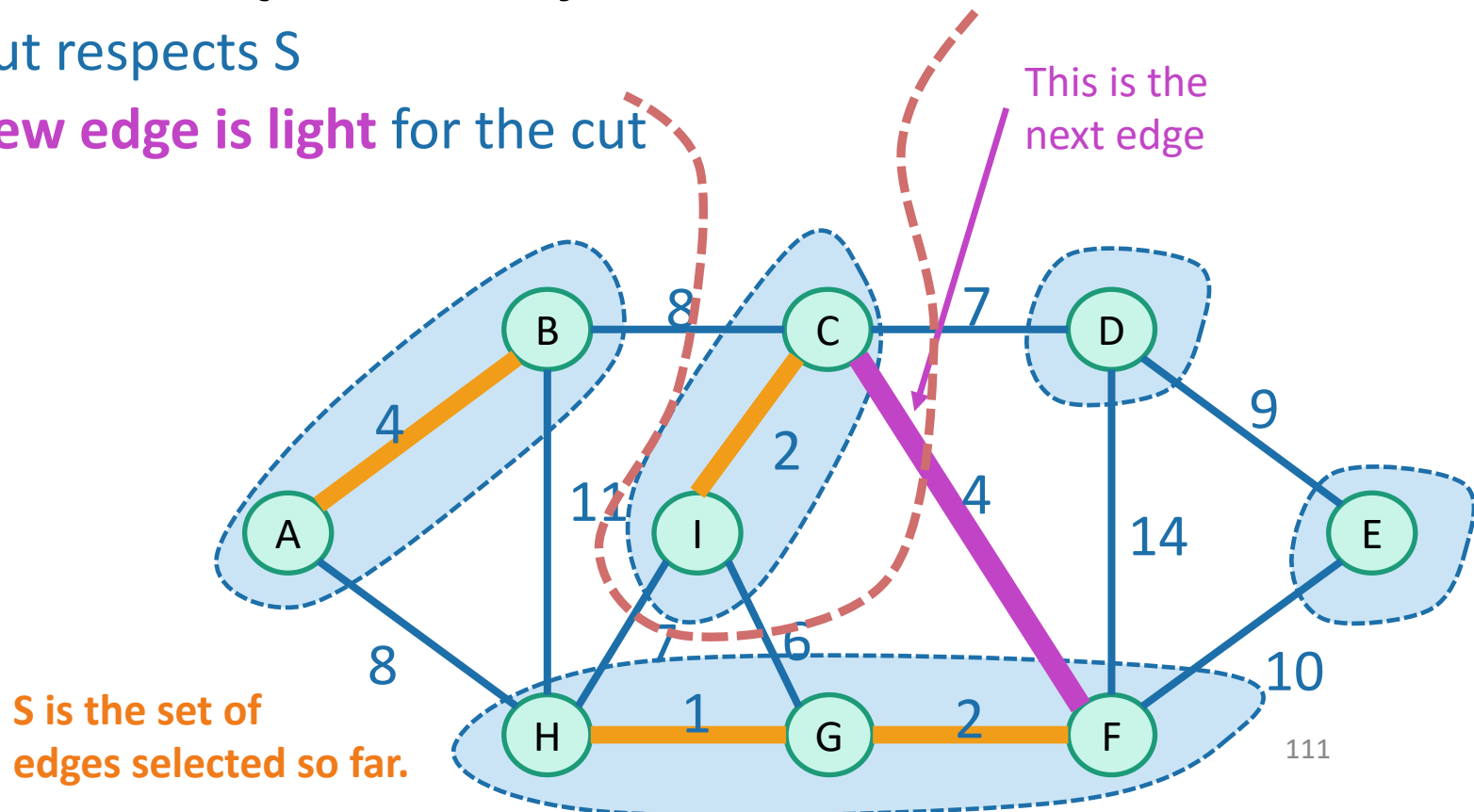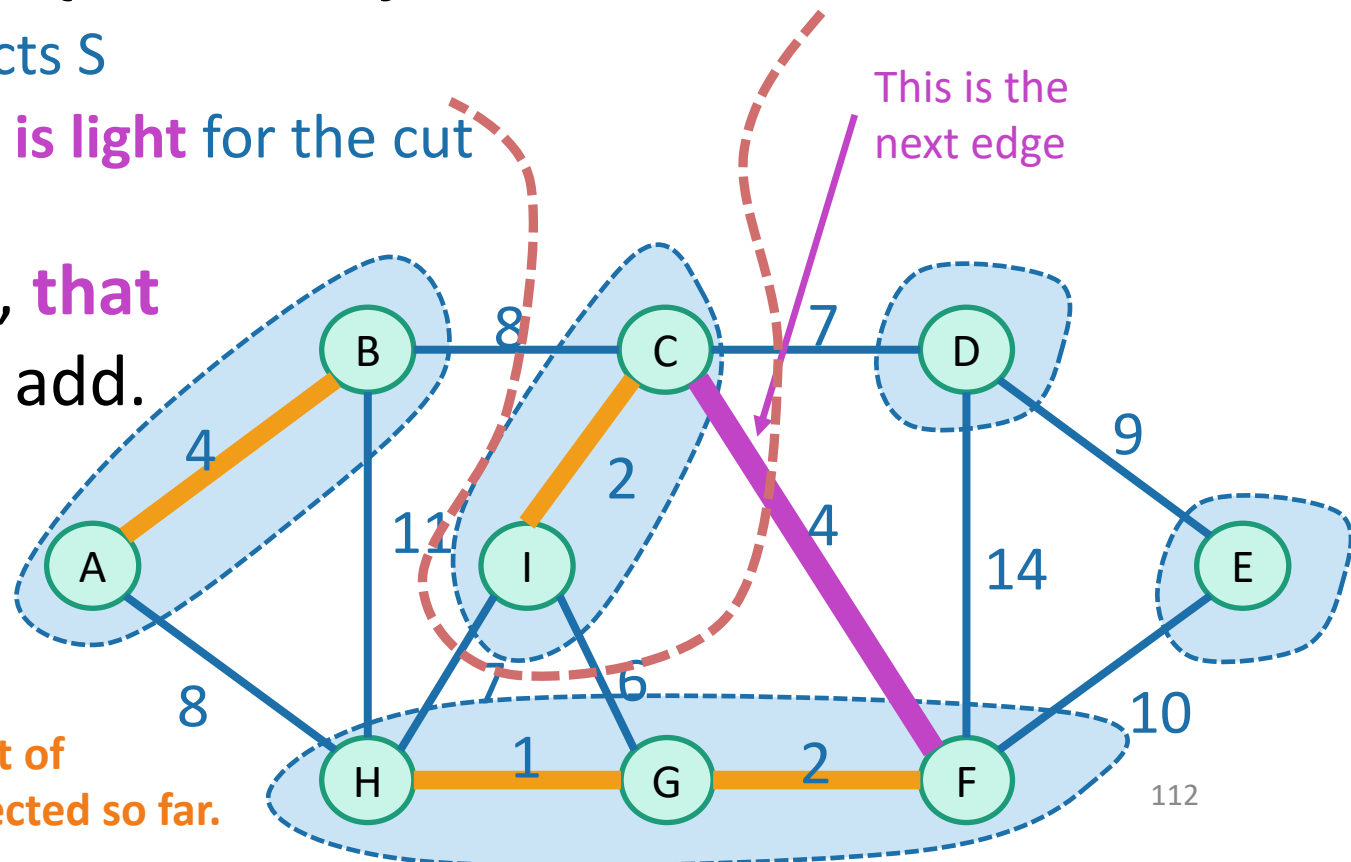- Assume that our choices **S** so far don't rule out success.
  - There is an MST extending them
- The **next edge** we add will merge two trees, **T1, T2**
- Consider the cut **{T1, V − T1}.**
  - This cut respects S
  - Our **new edge is light** for the cut

- By the Lemma, **that edge** is safe to add.
  - There is still an MST extending the new set

This is the next edge

S is the set of edges selected so far.

So

- Our greedy choices **don't rule out success**.

- This is enough (along with an argument by induction) to guarantee correctness of Kruskal's algorithm.

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
    - if the edge weights are reasonably-sized integers and we ignore the inverse Ackerman function, basically O(m) in practice.

- 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.**
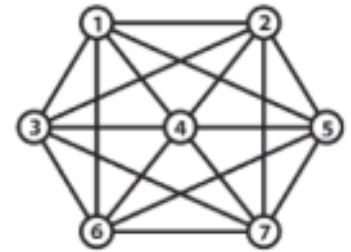
Compare and contrast

- Prim:
  - Grows a tree.
  - Time $O(m\log(n))$ with a binary-min-heap
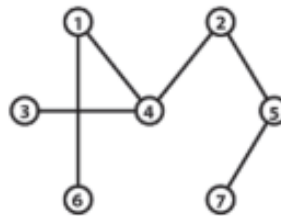 - Time $O(m + n\log(n))$ with a Fibonacci heap
- Kruskal:
  - Grows a forest.
  - Time $O(m\log(n))$ with a union-find data structure
  - If you can do radixSort on the edge weights, morally $O(m)$

Prim might be a better idea on dense graphs if you can't radixSort edge weights
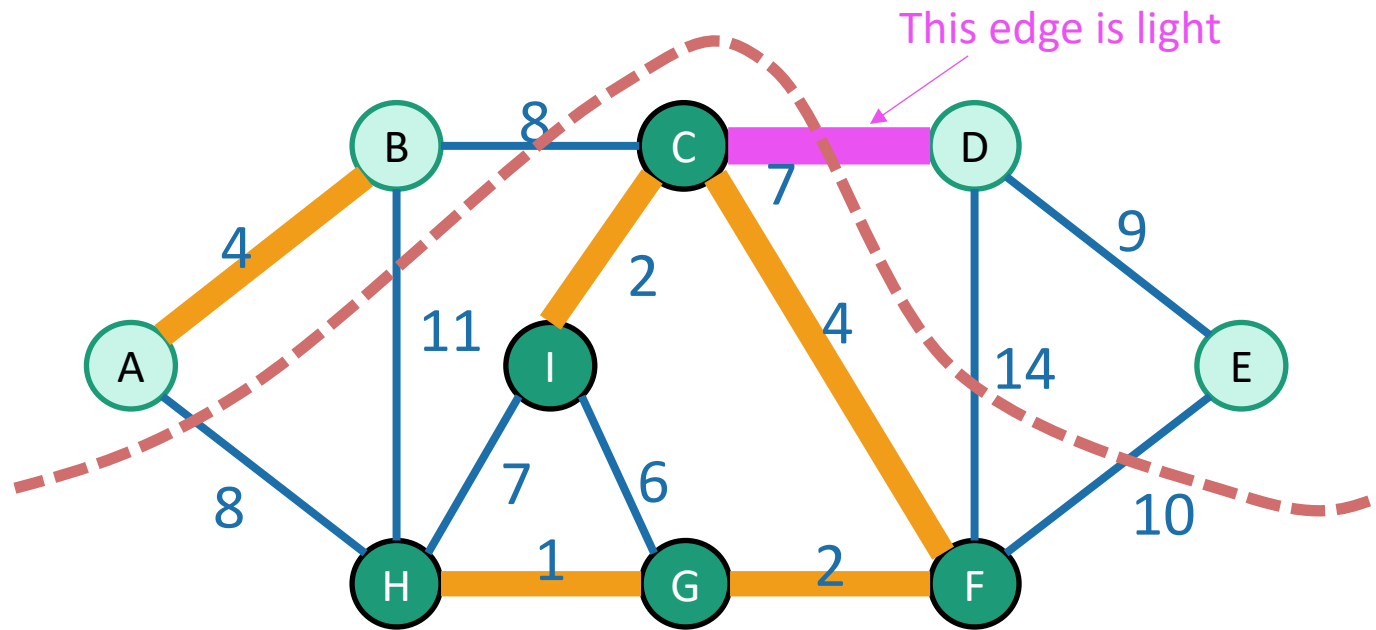

Dense

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

Sparse

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.

This edge is light

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

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**.