# Algorithms – I (CS29003/203)

Autumn 2022, IIT Kharagpur

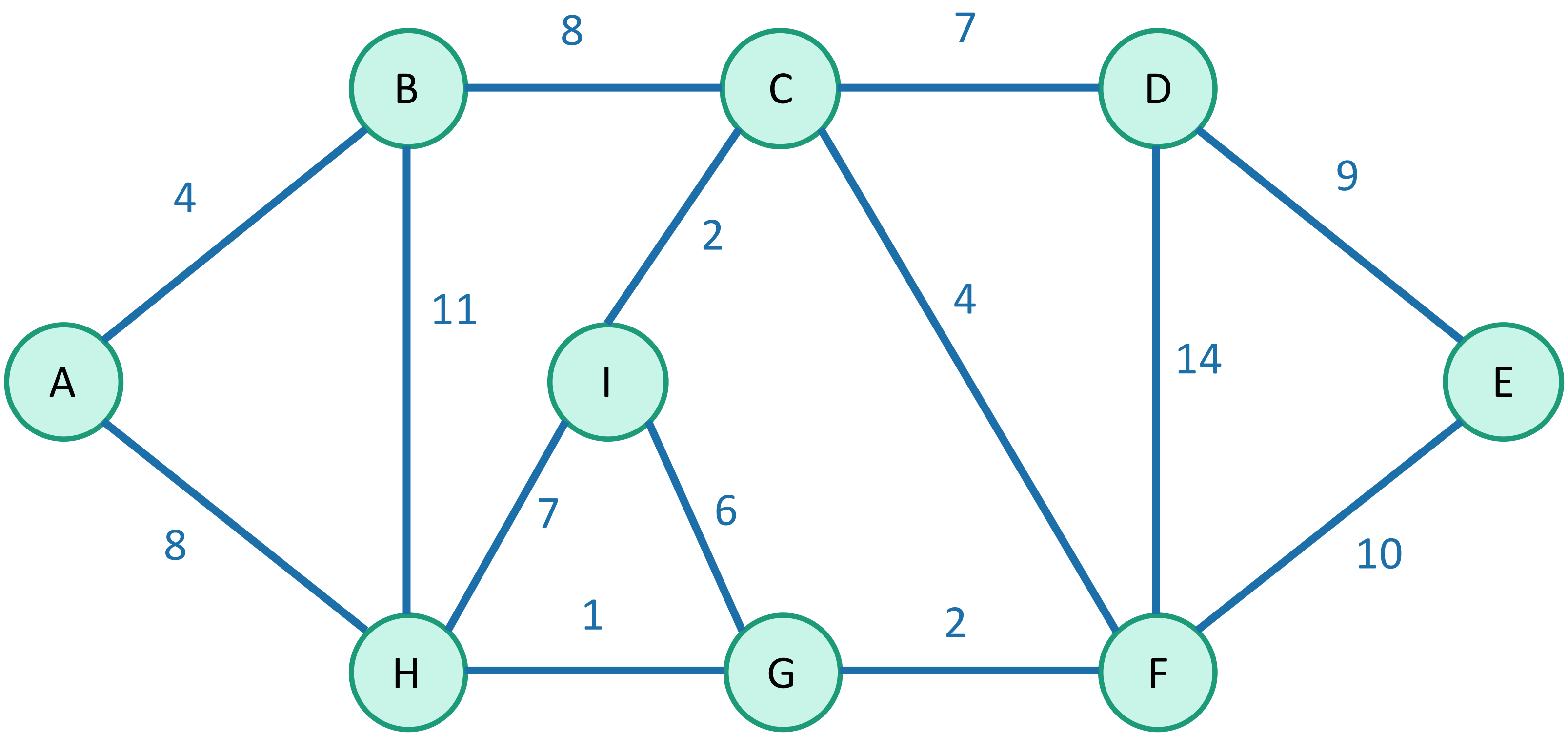
## Graphs

# Resources

- Apart from the book
- UC Riverside, CS 141 course, Fall 2021 by Prof. Yan Gu and Prof. Yihan Sun
- Stanford University, CS 161 course, Winter 2022 by Prof. Moses Charikar and Prof. Nima Anari

# Minimum Spanning Tree

- For today we will focus on connected graphs. Say we have an undirected weighted graph




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

*Source: Stanford, CS 161 course, Winter 2022*

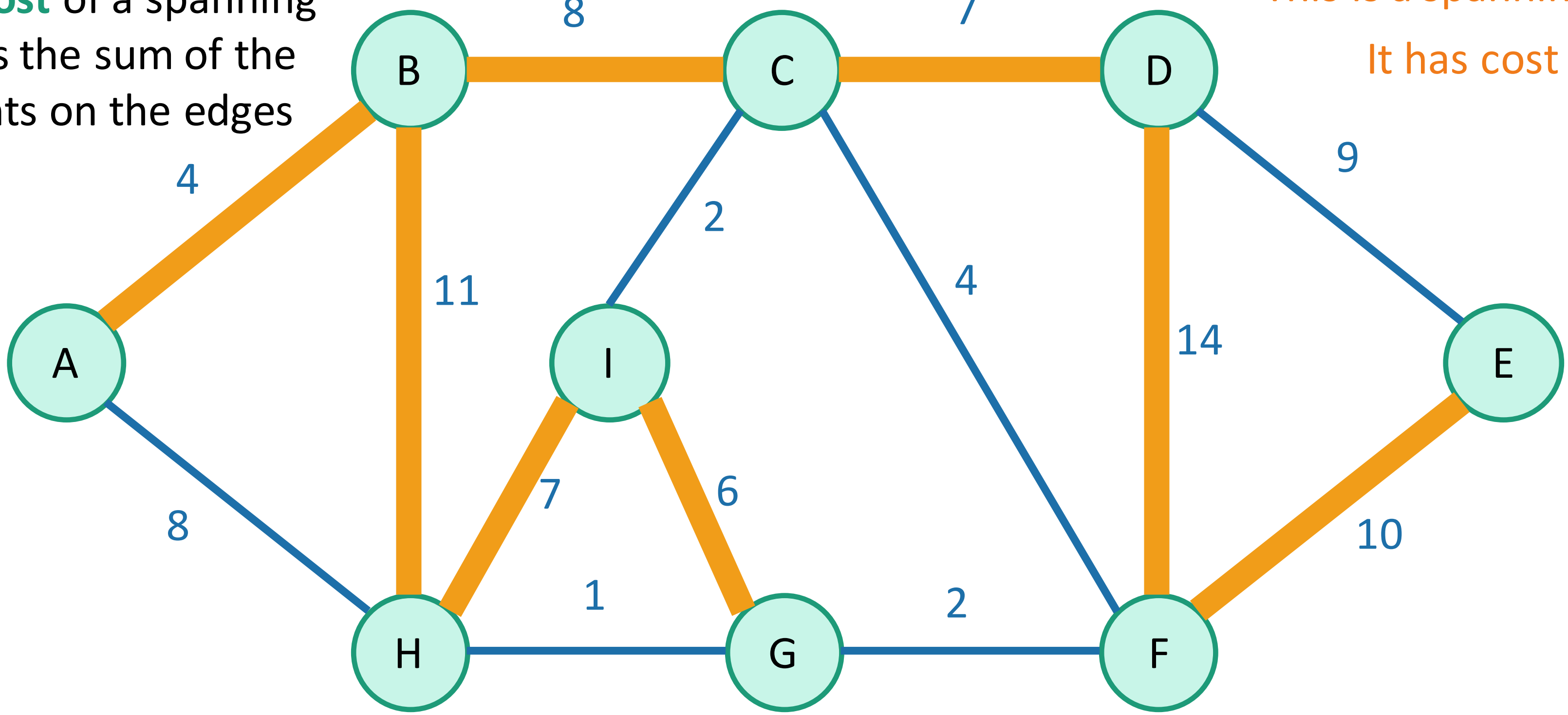
# Minimum Spanning Tree

- For today we will focus on connected graphs. 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

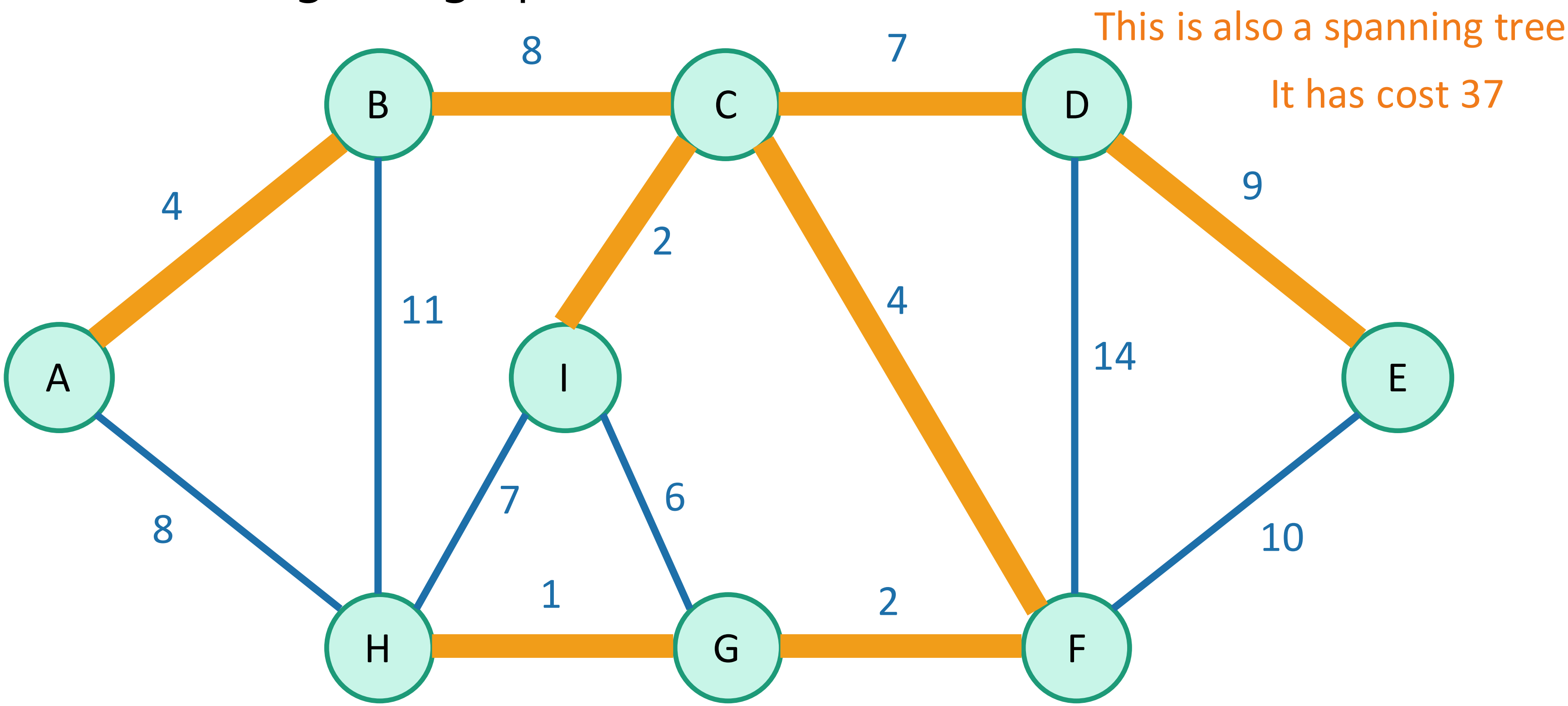



- A **spanning tree** is a **tree** that connects all of the vertices

Source: Stanford, CS 161 course, Winter 2022

# Minimum Spanning Tree

- For today we will focus on connected graphs. Say we have an undirected weighted graph

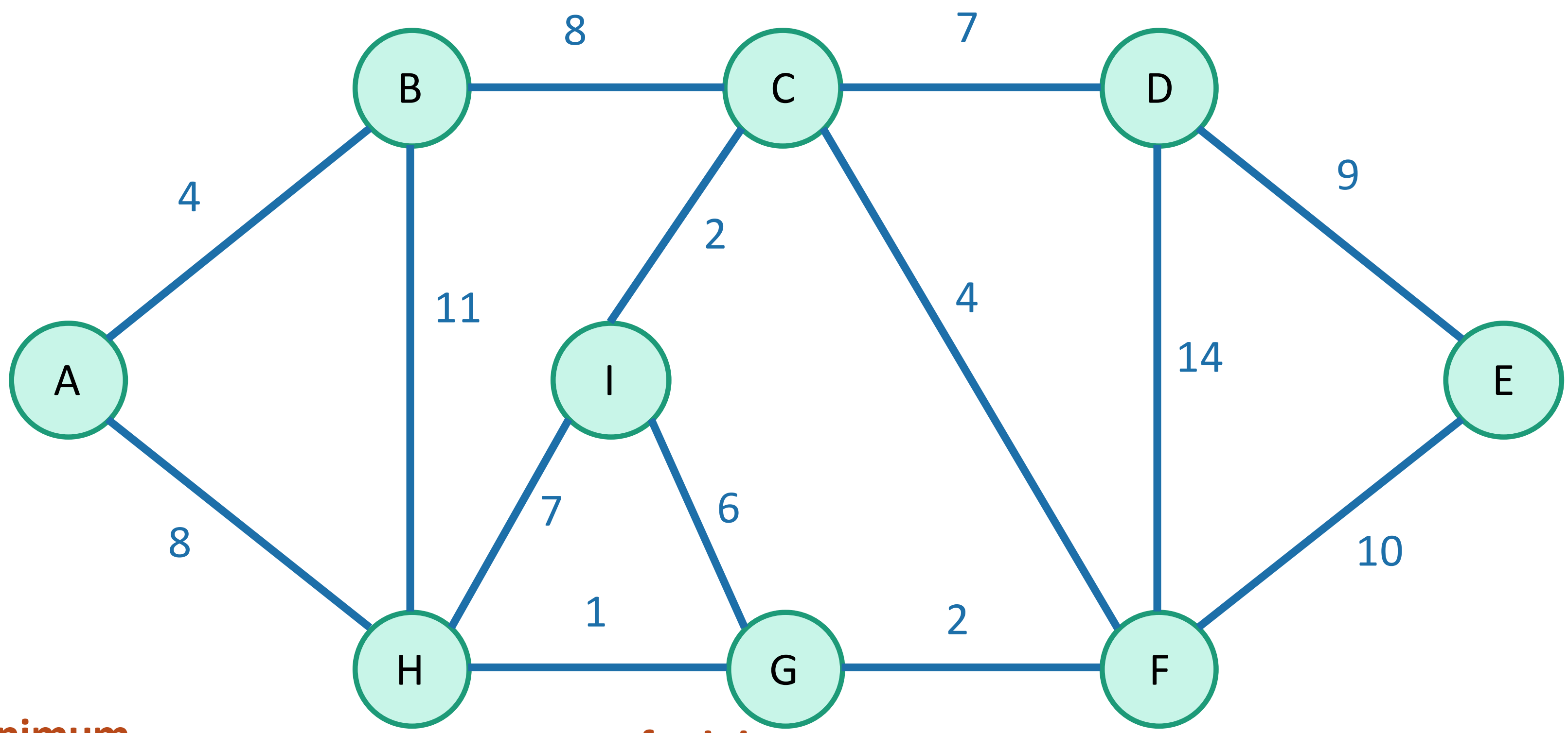



- A **spanning tree** is a **tree** that connects all of the vertices

*Source: Stanford, CS 161 course, Winter 2022*

# Minimum Spanning Tree

- For today we will focus on connected graphs. Say we have an undirected weighted graph




minimum

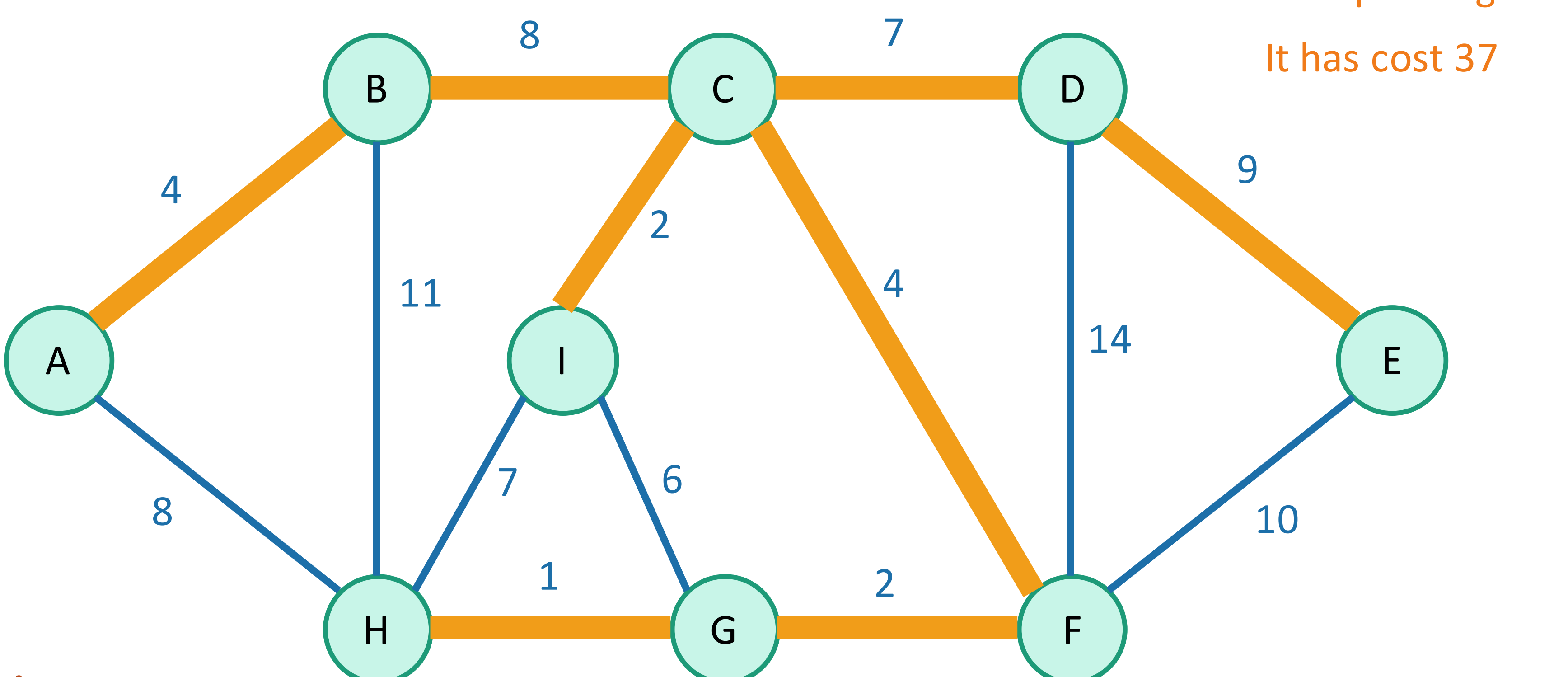
of minimum cost

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

*Source: Stanford, CS 161 course, Winter 2022*

# Minimum Spanning Tree

- For today we will focus on connected graphs. Say we have an undirected weighted graph



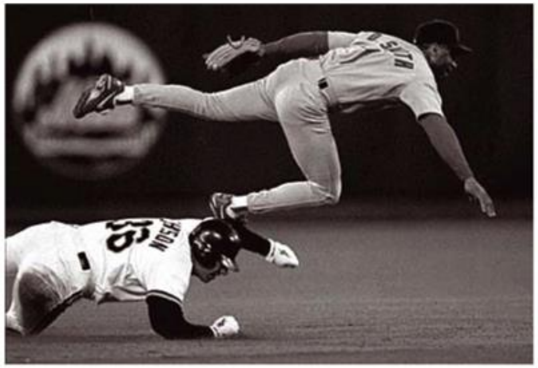

minimum
of minimum cost

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

*Source: Stanford, CS 161 course, Winter 2022*

# Why Minimum Spanning Trees

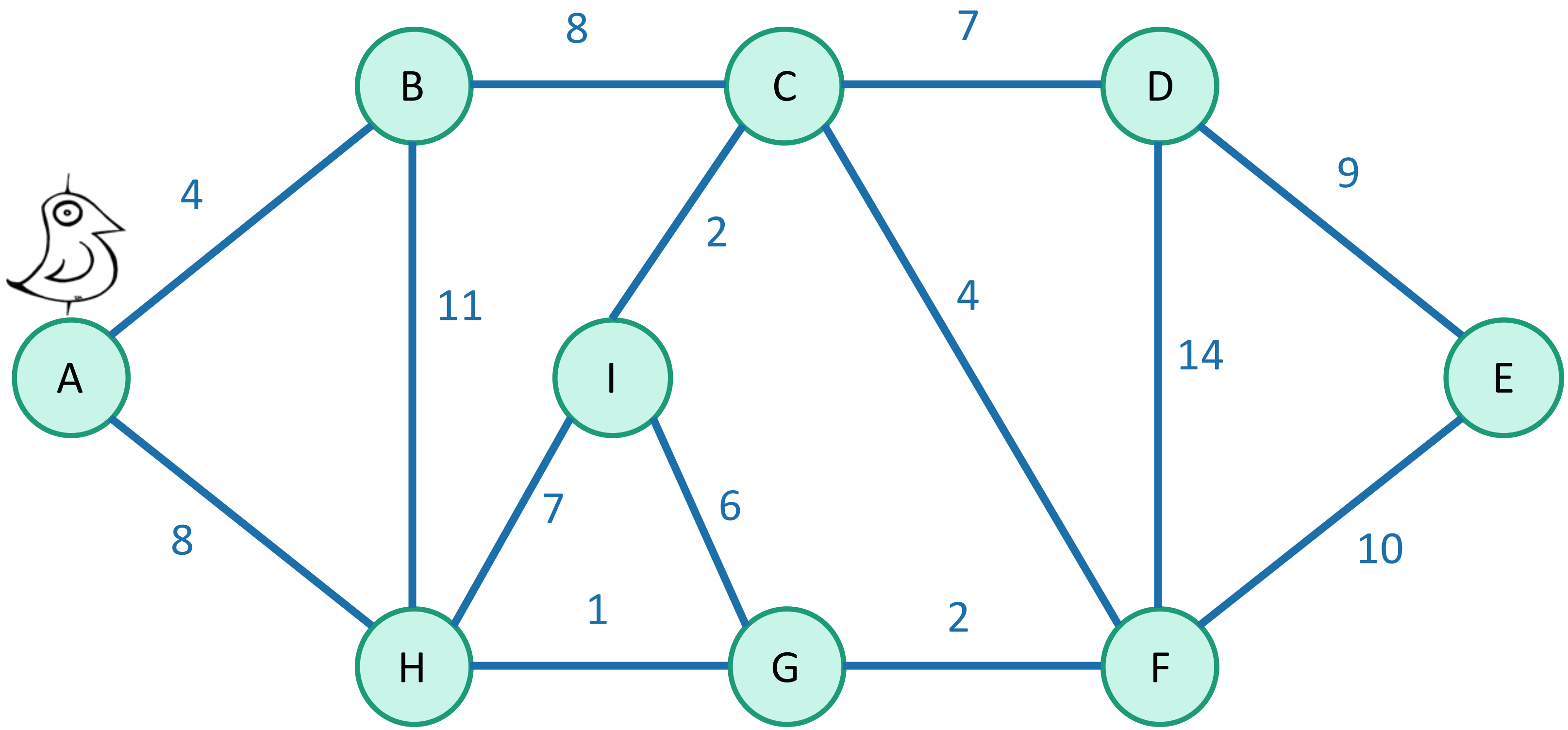
- Network Design
  - Connecting cities with roads/electricity/telephone/…
- Cluster analysis
  - e.g., genetic distance
- Image processing
  - e.g., image segmentation
- Useful primitive
  - for other graph algorithms



*Ref: Felzenswalb et. Al., Efficient graph-based image segmentation, IJCV 2004*

# How to find Minimum Spanning Trees

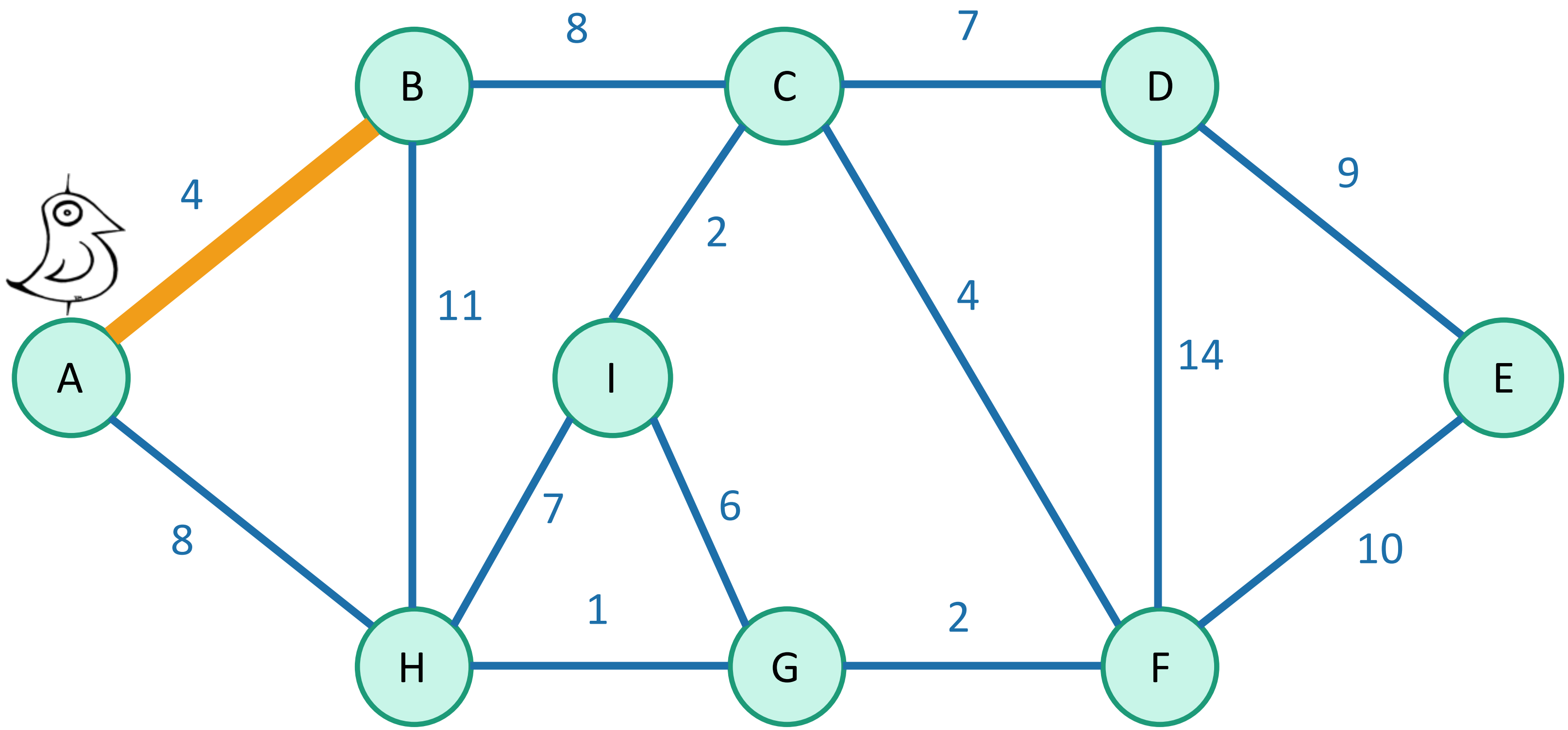
- We will see two greedy algorithms
- Start growing a tree, greedily add the shortest edge we can to grow the tree




Source: Stanford, CS 161 course, Winter 2022

# How to find Minimum Spanning Trees

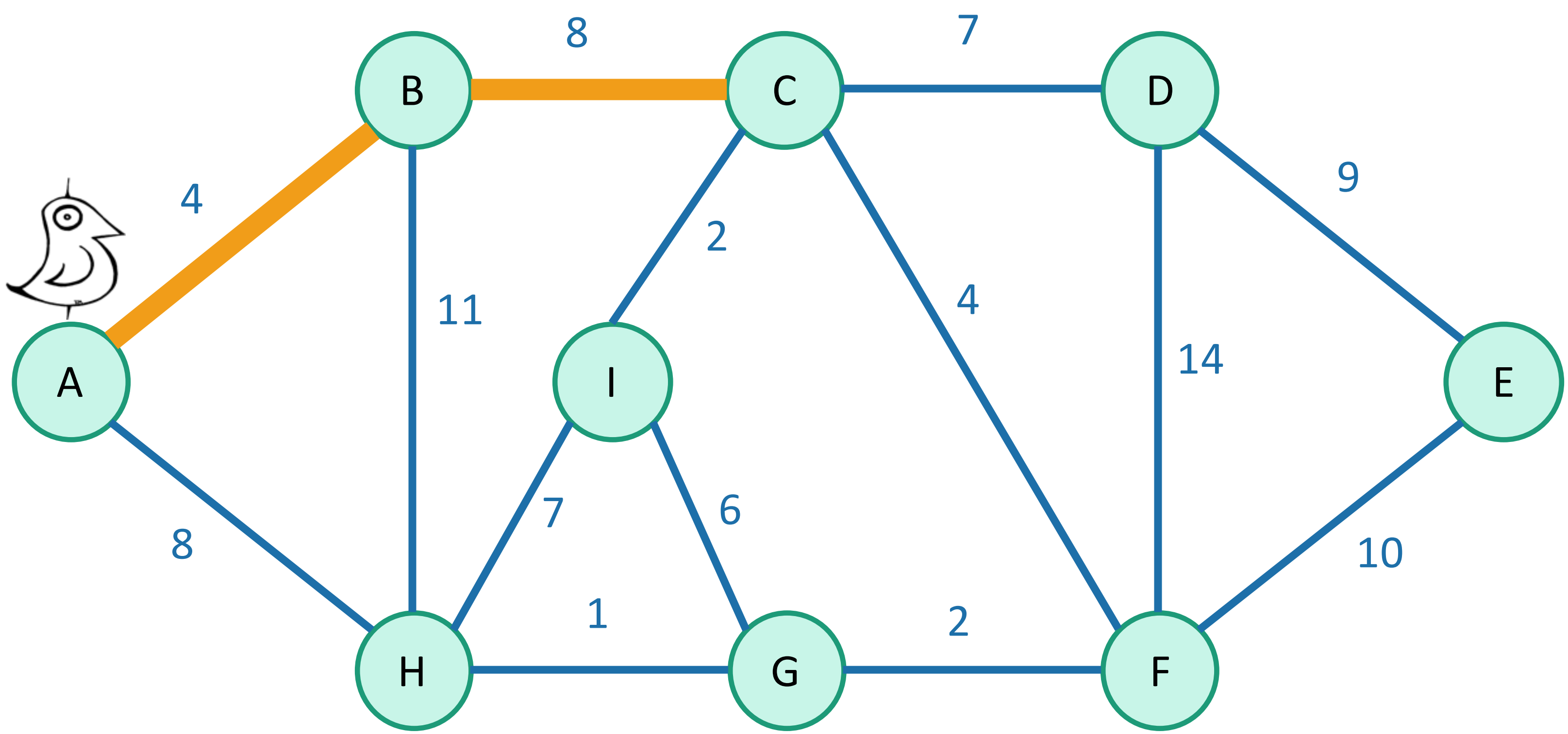
- We will see two greedy algorithms
- Start growing a tree, greedily add the shortest edge we can to grow the tree




Source: Stanford, CS 161 course, Winter 2022

# How to find Minimum Spanning Trees

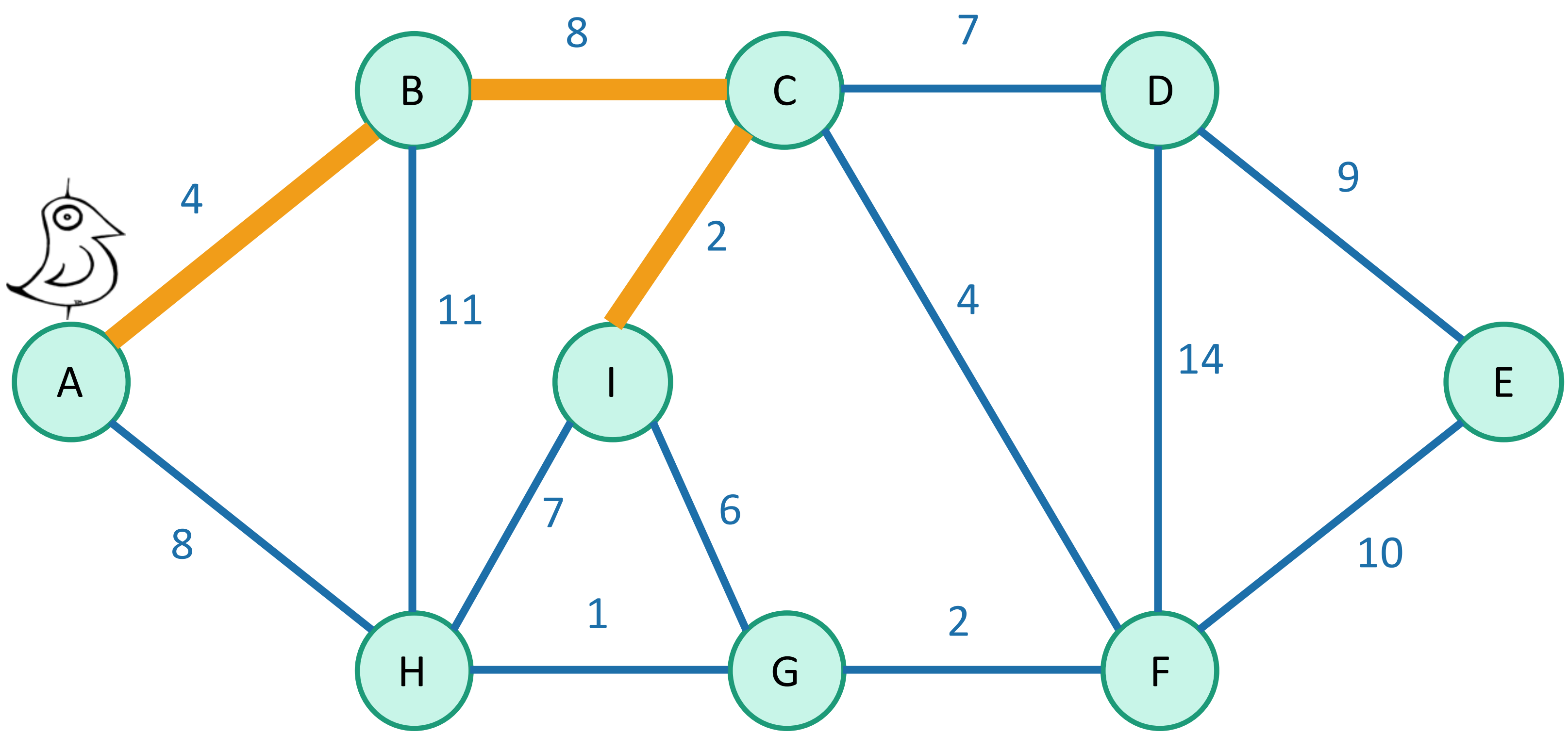
- We will see two greedy algorithms
- Start growing a tree, greedily add the shortest edge we can to grow the tree




*Source: Stanford, CS 161 course, Winter 2022*

# How to find Minimum Spanning Trees

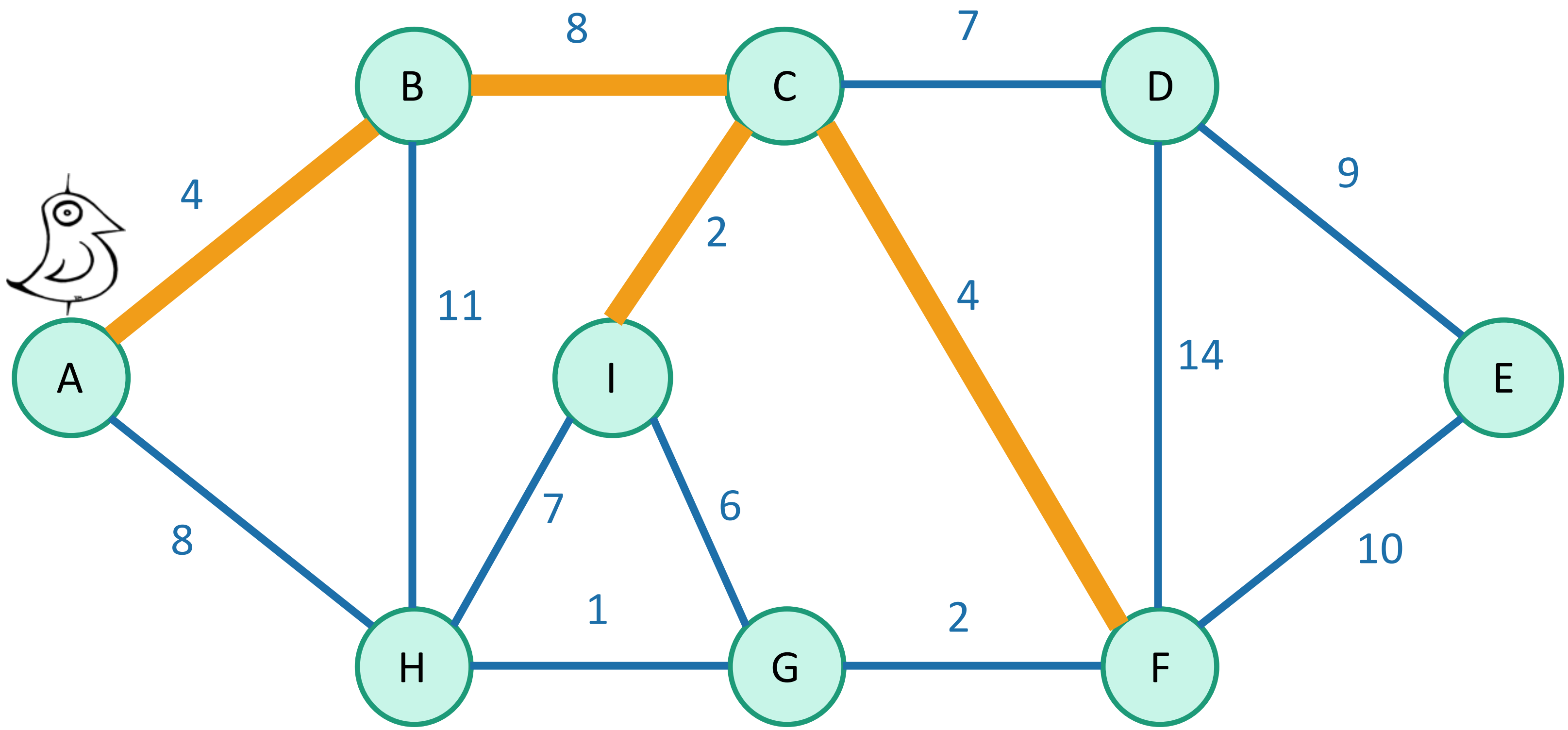
- We will see two greedy algorithms
- Start growing a tree, greedily add the shortest edge we can to grow the tree




Source: Stanford, CS 161 course, Winter 2022

# How to find Minimum Spanning Trees

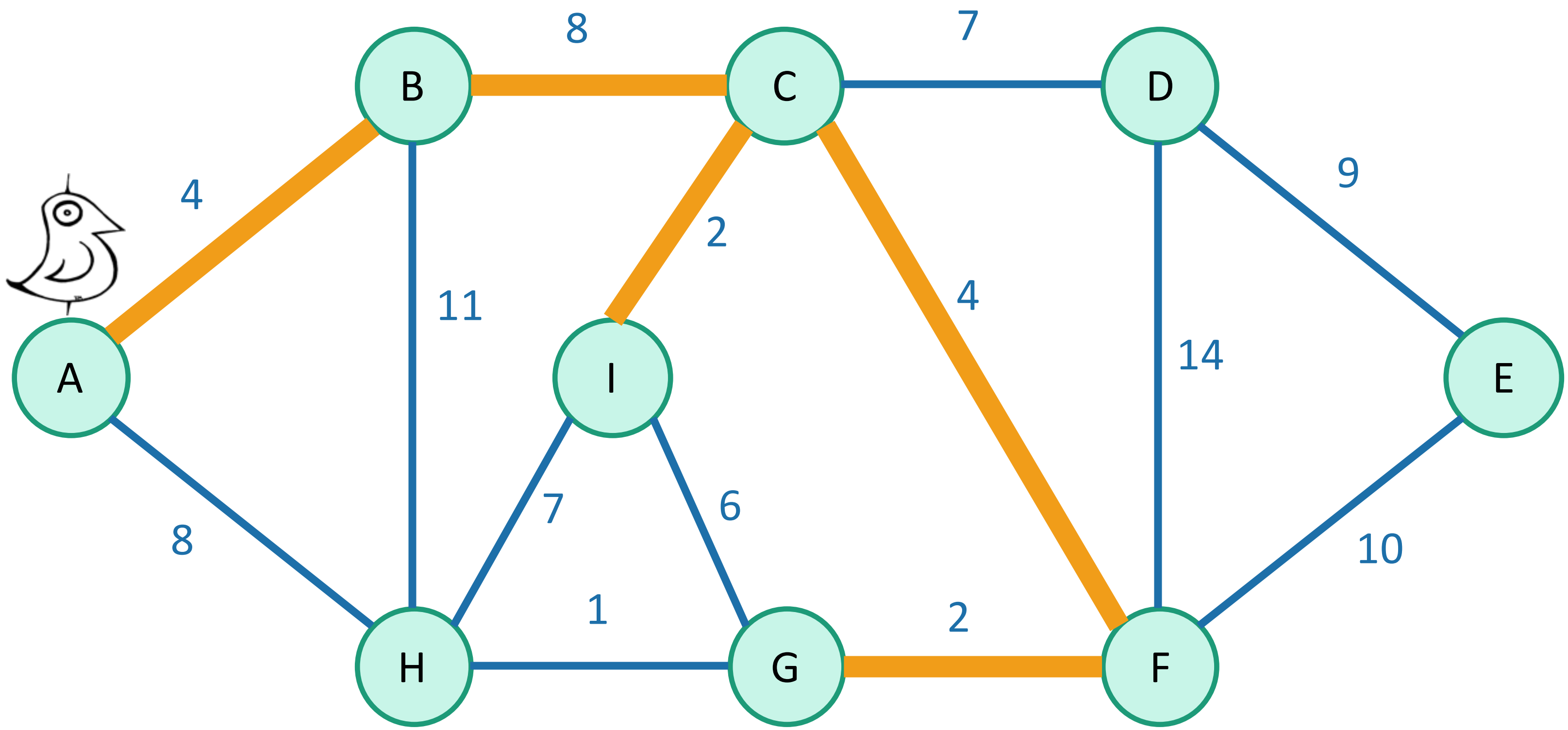
- We will see two greedy algorithms
- Start growing a tree, greedily add the shortest edge we can to grow the tree




Source: Stanford, CS 161 course, Winter 2022

# How to find Minimum Spanning Trees

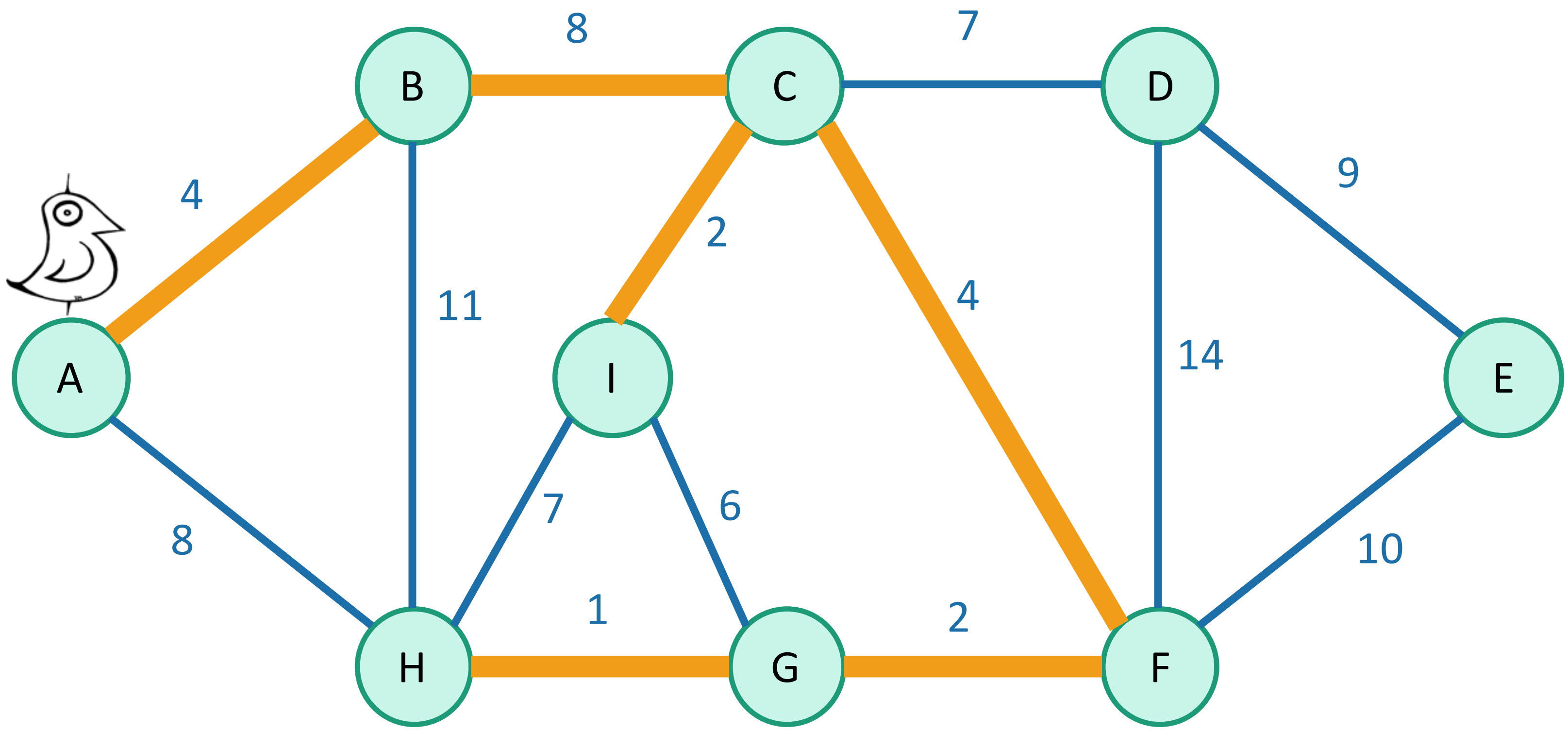
- We will see two greedy algorithms
- Start growing a tree, greedily add the shortest edge we can to grow the tree




Source: Stanford, CS 161 course, Winter 2022

# How to find Minimum Spanning Trees

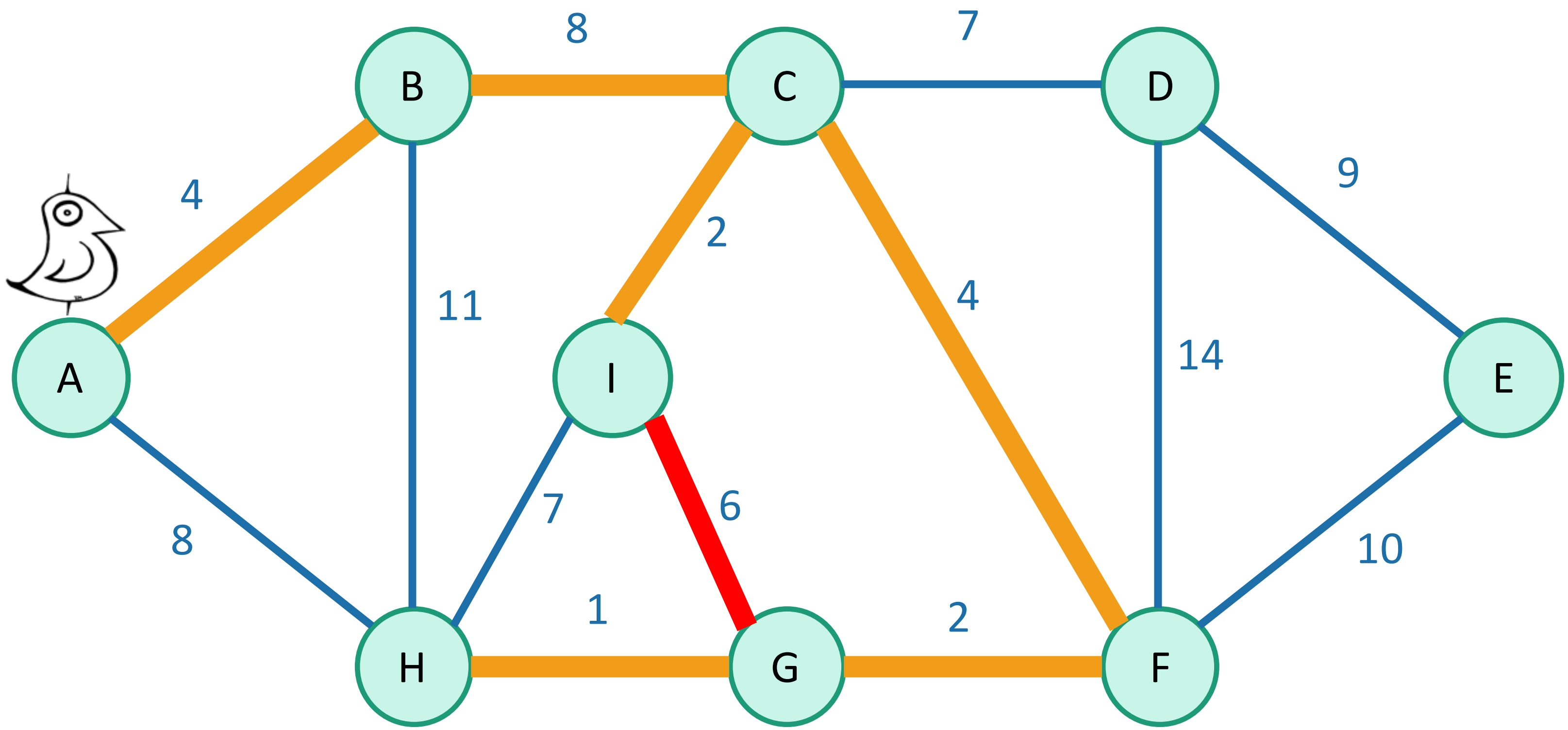
- We will see two greedy algorithms
- Start growing a tree, greedily add the shortest edge we can to grow the tree




Source: Stanford, CS 161 course, Winter 2022

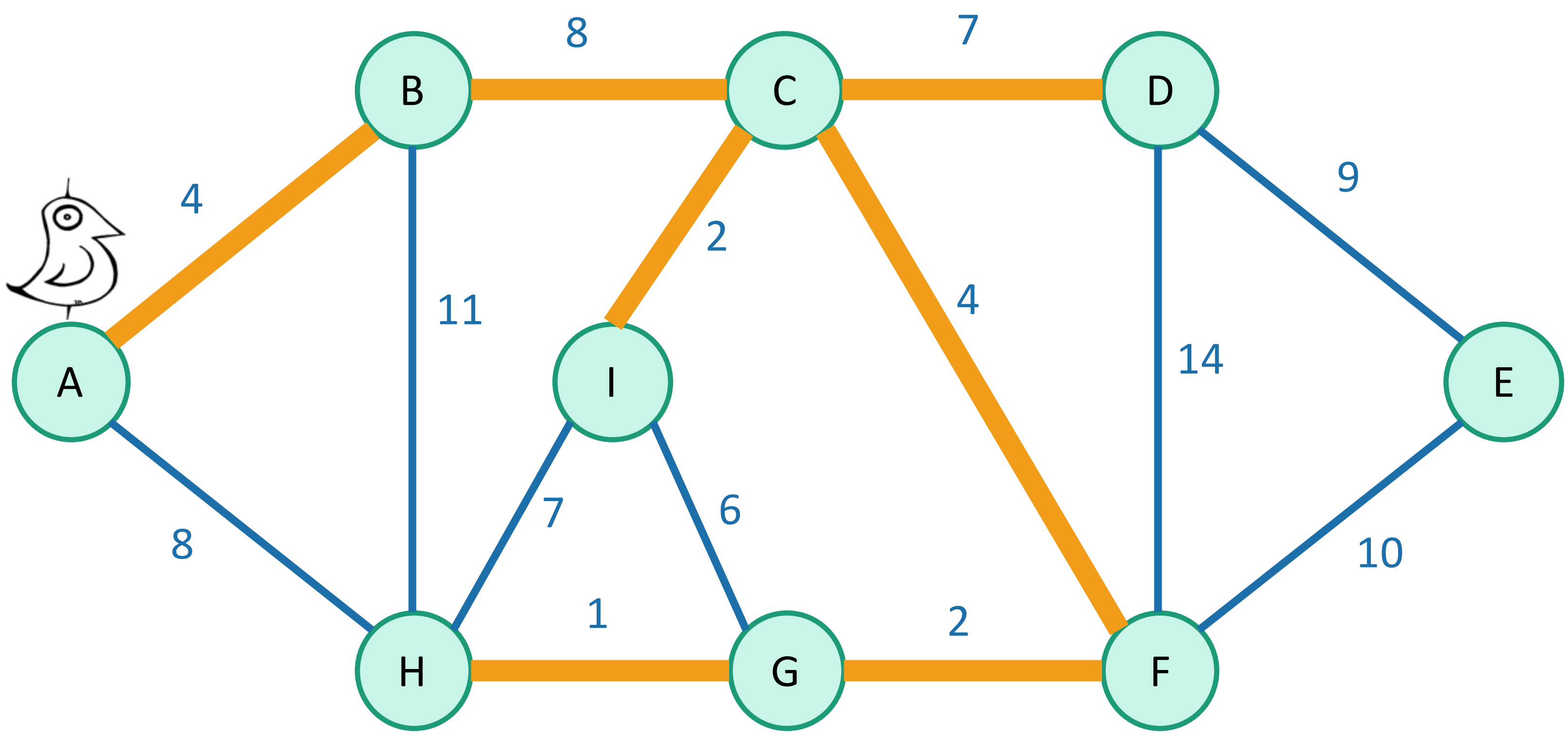
# How to find Minimum Spanning Trees

- We will see two greedy algorithms
- Start growing a tree, greedily add the shortest edge we can to grow the tree



# How to find Minimum Spanning Trees

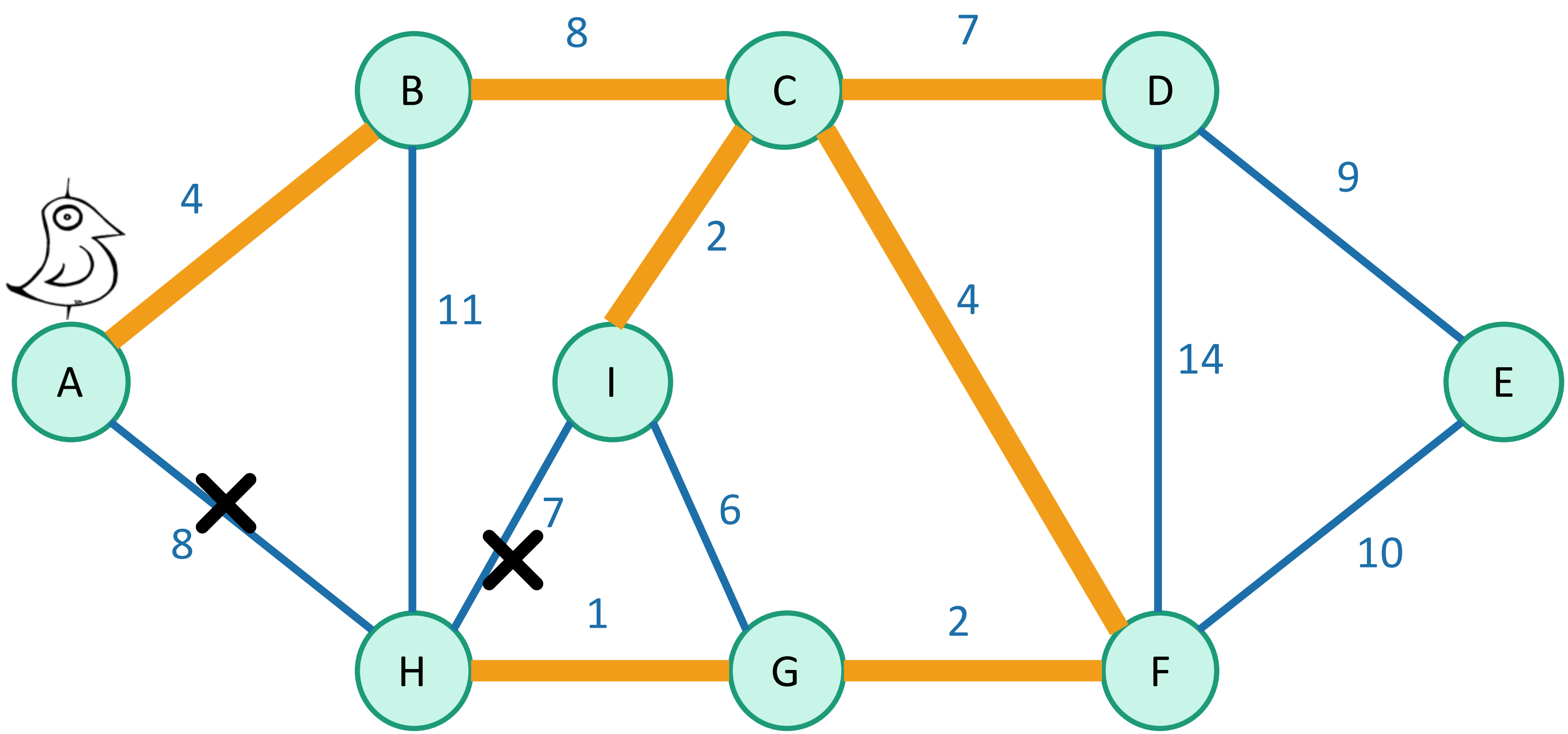
- We will see two greedy algorithms
- Start growing a tree, greedily add the shortest edge we can to grow the tree




Source: Stanford, CS 161 course, Winter 2022

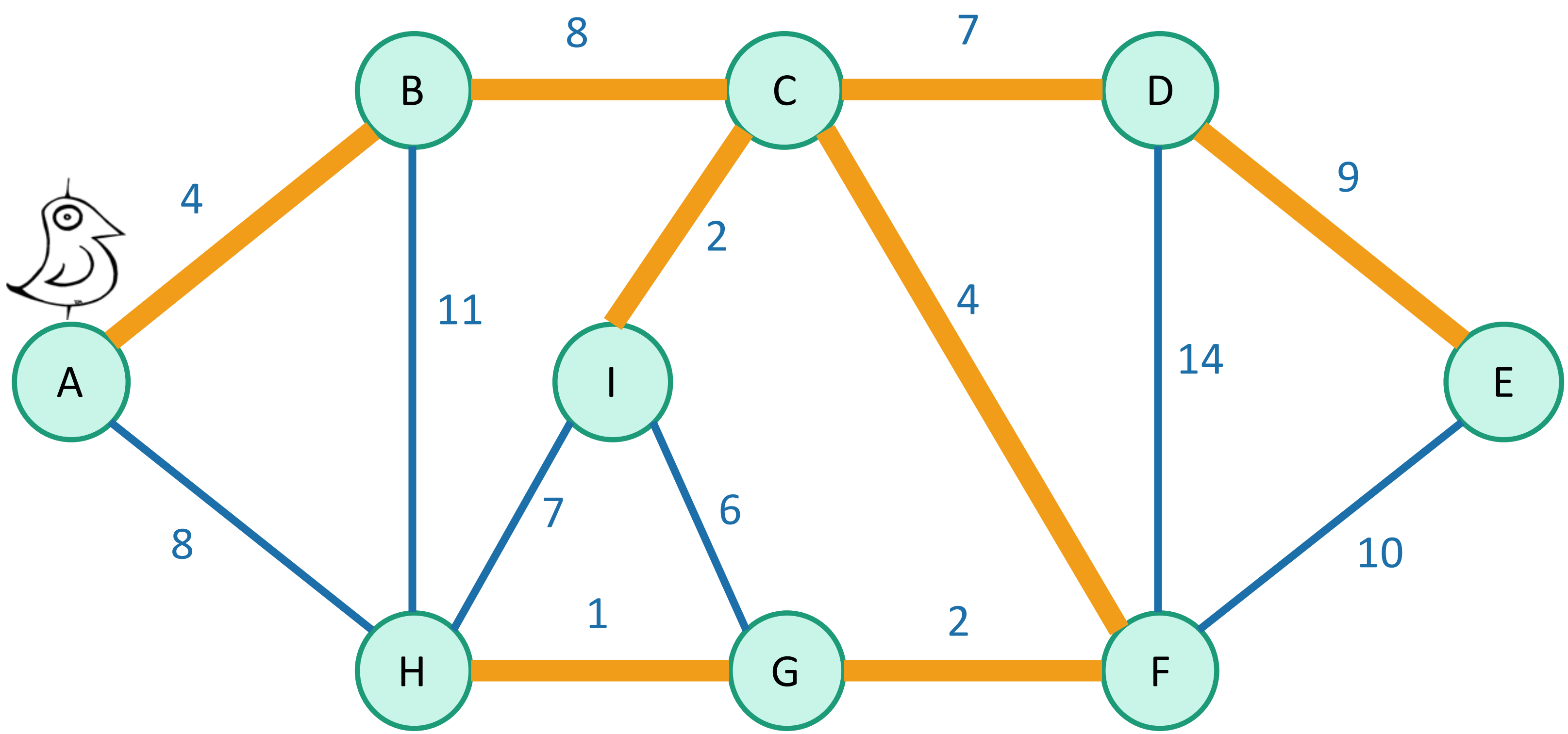
# How to find Minimum Spanning Trees

- We will see two greedy algorithms
- Start growing a tree, greedily add the shortest edge we can to grow the tree



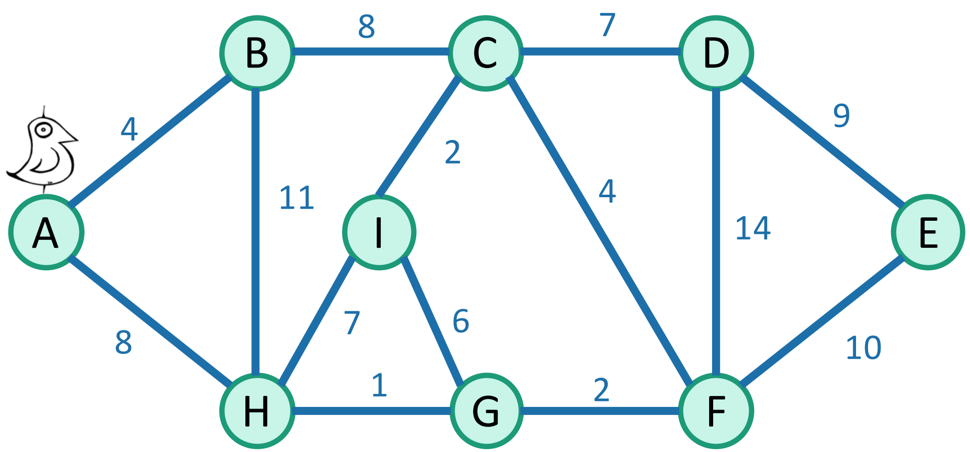
# How to find Minimum Spanning Trees

- We will see two greedy algorithms
- Start growing a tree, greedily add the shortest edge we can to grow the tree



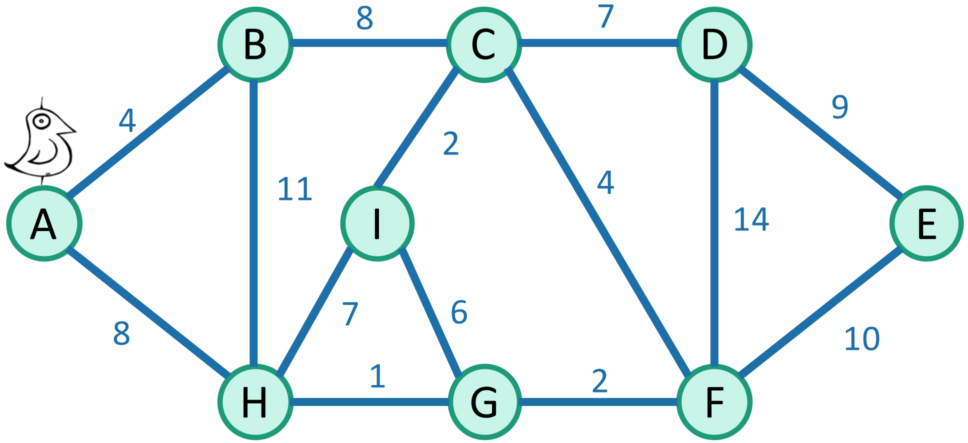

Source: Stanford, CS 161 course, Winter 2022

# How to find Minimum Spanning Trees




| | key | $\pi$ |
|---|---|---|
| A | 0 | Null |
| B | ∞ | Null |
| C | ∞ | Null |
| D | ∞ | Null |
| E | ∞ | Null |
| F | ∞ | Null |
| G | ∞ | Null |
| H | ∞ | Null |
| I | ∞ | Null |

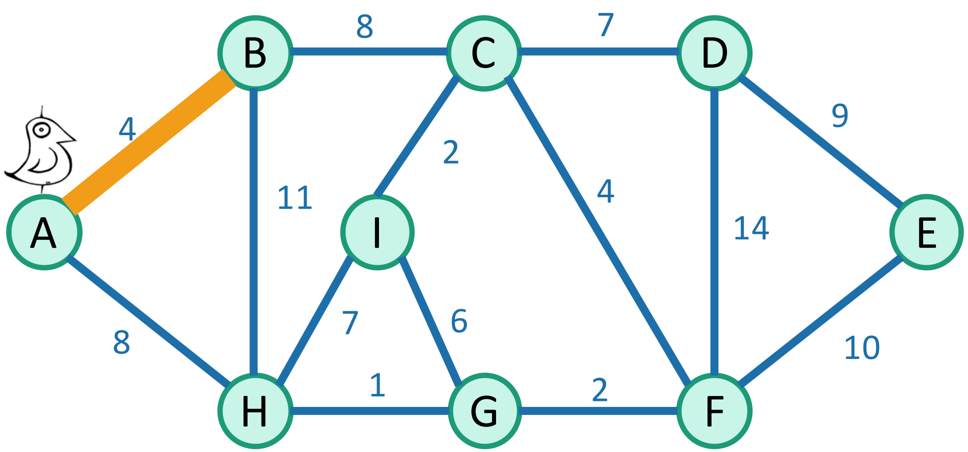
# How to find Minimum Spanning Trees




| | key | $\pi$ |
|---|---|---|
| A | 0 | Null |
| B | ~~∞~~ 4 | ~~Null~~ A |
| C | ∞ | Null |
| D | ∞ | Null |
| E | ∞ | Null |
| F | ∞ | Null |
| G | ∞ | Null |
| H | ~~∞~~ 8 | ~~Null~~ A |
| I | ∞ | Null |

- Now $A$ is out
- Also, $AB$ is the greedy choice. So, I shall first update the neighbors of $B$ and I shall not consider $B$ again
- Note that $A$'s neighbors are already in updated state in the priority queue

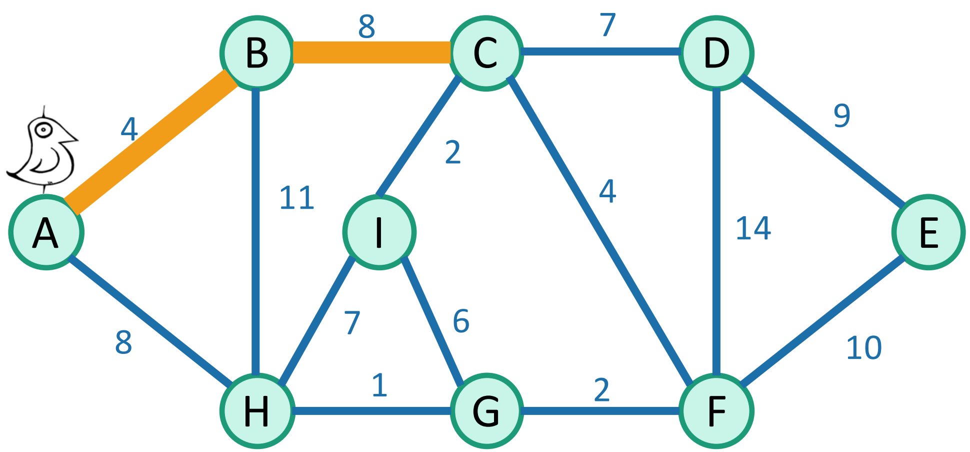
# How to find Minimum Spanning Trees




| | key | $\pi$ |
|---|---|---|
| A | 0 | Null |
| B | ~~∞~~ 4 | ~~Null~~ A |
| C | ~~∞~~ 8 | ~~Null~~ B |
| D | ∞ | Null |
| E | ∞ | Null |
| F | ∞ | Null |
| G | ∞ | Null |
| H | ~~∞~~ 8 11 | ~~Null~~ A B |
| I | ∞ | Null |

- Now $A$ is out
- Also, $AB$ is the greedy choice. So, I shall first update the neighbors of $B$ and I shall not consider $B$ again
- Note that $A$'s neighbors are already in updated state in the priority queue

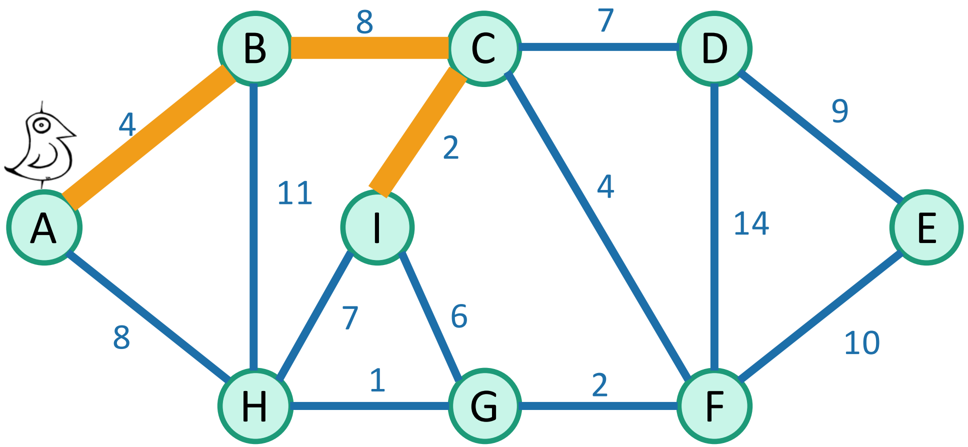
# How to find Minimum Spanning Trees




| | key | $\pi$ |
|---|---|---|
| A | 0 | Null |
| B | ~~∞~~ 4 | ~~Null~~ A |
| C | ~~∞~~ 8 | ~~Null~~ B |
| D | ~~∞~~ 7 | ~~Null~~ C |
| E | ∞ | Null |
| F | ~~∞~~ 4 | ~~Null~~ C |
| G | ∞ | Null |
| H | ~~∞~~ 8 | ~~Null~~ A |
| I | ~~∞~~ 2 | ~~Null~~ C |

- Now $B$ is out
- Also, $BC$ is the greedy choice. So, I shall first update the neighbors of $C$ and I shall not consider $C$ again

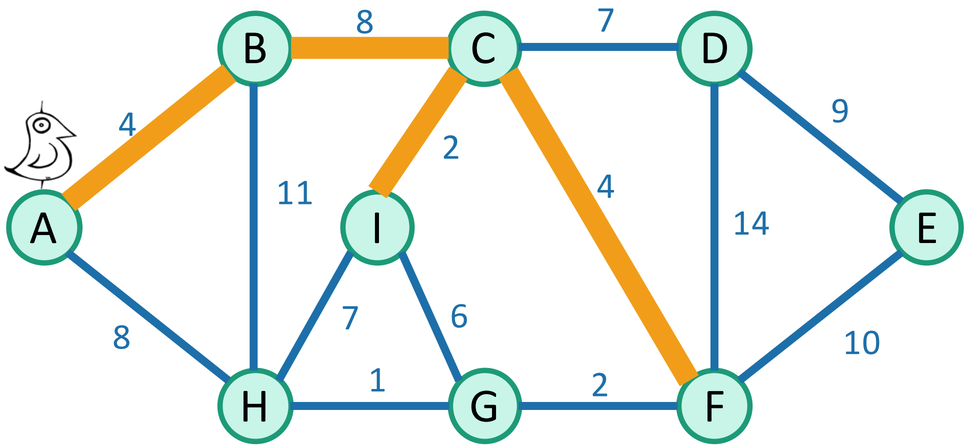
# How to find Minimum Spanning Trees




| | key | $\pi$ |
|---|---|---|
| ☑ A | 0 | Null |
| ☑ B | ~~∞~~ 4 | ~~Null~~ A |
| ☑ C | ~~∞~~ 8 | ~~Null~~ B |
| D | ~~∞~~ 7 | ~~Null~~ C |
| E | ∞ | Null |
| F | ~~∞~~ 4 | ~~Null~~ C |
| G | ~~∞~~ 6 | ~~Null~~ I |
| H | ~~∞~~ ~~8~~ 7 | ~~Null~~ ~~A~~ I |
| ☑ I | ~~∞~~ 2 | ~~Null~~ C |

- Now $C$ is out
- Also, $CI$ is the greedy choice. So, I shall first update the neighbors of $I$ and I shall not consider $I$ again

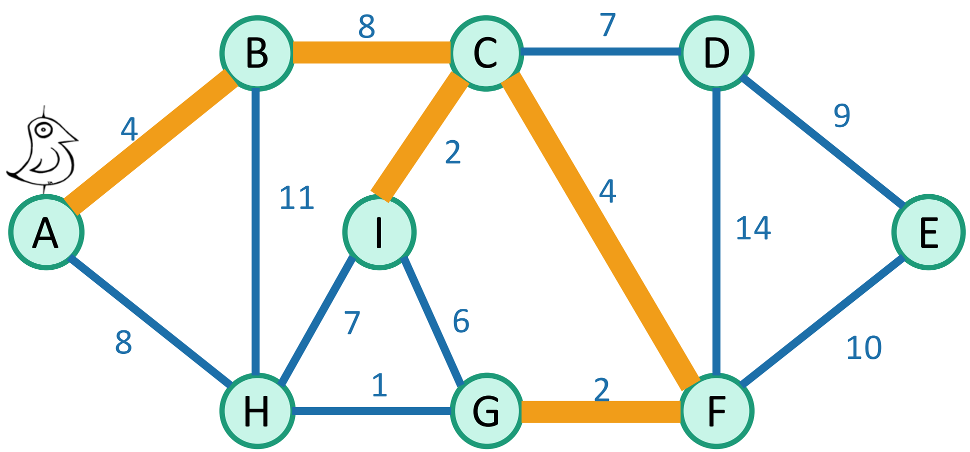
# How to find Minimum Spanning Trees




| | key | $\pi$ |
|---|---|---|
| ☑ A | 0 | Null |
| ☑ B | ~~∞~~ 4 | ~~Null~~ A |
| ☑ C | ~~∞~~ 8 | ~~Null~~ B |
| D | ~~∞~~ 7 | ~~Null~~ C |
| E | ~~∞~~ 10 | ~~Null~~ F |
| ☑ F | ~~∞~~ 4 | ~~Null~~ C |
| G | ~~∞~~ ~~6~~ 2 | ~~Null~~ ~~I~~ F |
| H | ~~∞~~ ~~8~~ 7 | ~~Null~~ ~~A~~ I |
| ☑ I | ~~∞~~ 2 | ~~Null~~ C |

- Now $I$ is out
- Also, $CF$ is the greedy choice. So, I shall first update the neighbors of $F$ and I shall not consider $F$ again

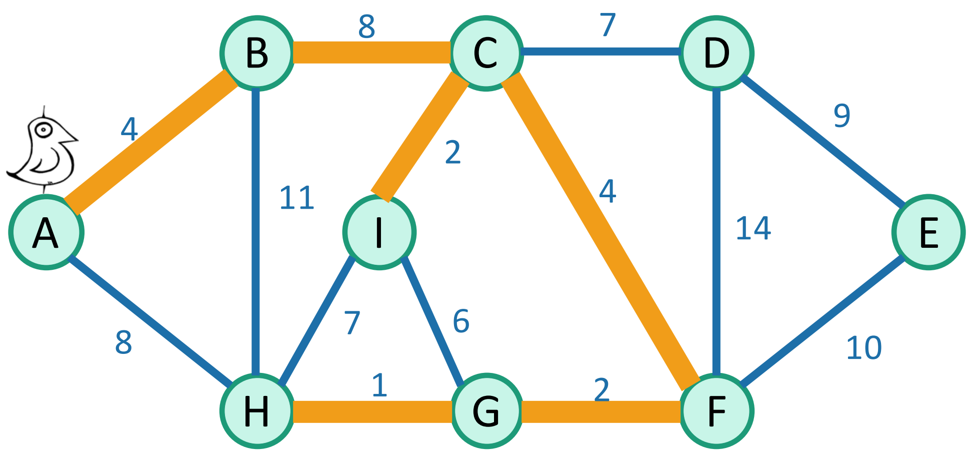
# How to find Minimum Spanning Trees




| | key | $\pi$ |
|---|---|---|
| ✓ A | 0 | Null |
| ✓ B | ~~∞~~ 4 | ~~Null~~ A |
| ✓ C | ~~∞~~ 8 | ~~Null~~ B |
| D | ~~∞~~ 7 | ~~Null~~ C |
| E | ~~∞~~ 10 | ~~Null~~ F |
| ✓ F | ~~∞~~ 4 | ~~Null~~ C |
| ✓ G | ~~∞~~ ~~6~~ 2 | ~~Null~~ ~~I~~ F |
| H | ~~∞~~ ~~8~~ ~~7~~ 1 | ~~Null~~ ~~A~~ ~~I~~ G |
| ✓ I | ~~∞~~ 2 | ~~Null~~ C |

- Now $F$ is out
- Also, $FG$ is the greedy choice. So, I shall first update the neighbors of $G$ and I shall not consider $G$ again

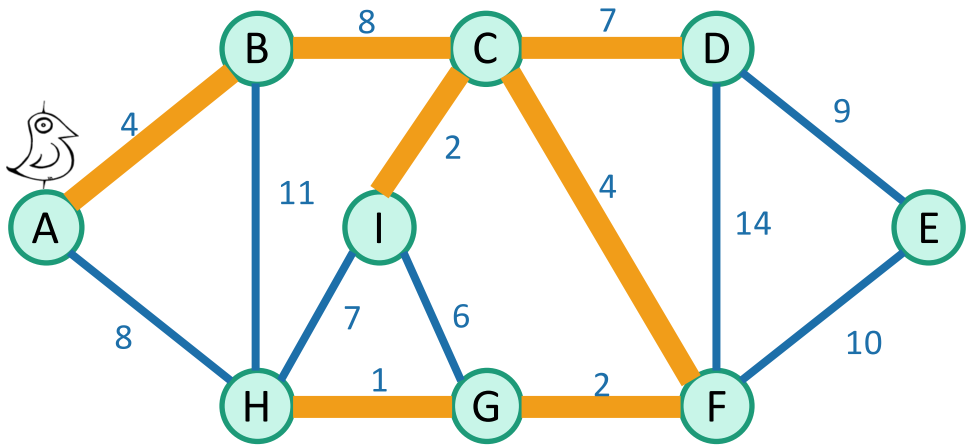
# How to find Minimum Spanning Trees




| | key | $\pi$ |
|---|---|---|
| ☑ A | 0 | Null |
| ☑ B | ~~∞~~ 4 | ~~Null~~ A |
| ☑ C | ~~∞~~ 8 | ~~Null~~ B |
| D | ~~∞~~ 7 | ~~Null~~ C |
| E | ~~∞~~ 10 | ~~Null~~ F |
| ☑ F | ~~∞~~ 4 | ~~Null~~ C |
| ☑ G | ~~∞~~ ~~6~~ 2 | ~~Null~~ ~~I~~ F |
| ☑ H | ~~∞~~ ~~8~~ ~~7~~ 1 | ~~Null~~ ~~A~~ ~~I~~ G |
| ☑ I | ~~∞~~ 2 | ~~Null~~ C |

- Now $G$ is out
- Also, $GH$ is the greedy choice. So, I shall first update the neighbors of $H$ and I shall not consider $H$ again
- In fact, all neighbors are out in this case – This also means a cycle would come. So we don't do anything
- Same scenario with next greedy choices, $GI, HI$

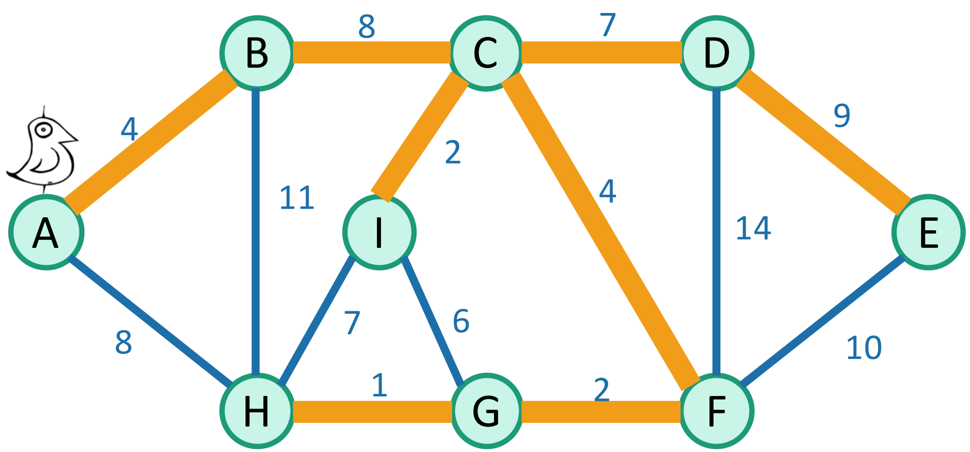
# How to find Minimum Spanning Trees




| | key | $\pi$ |
|---|---|---|
| ✓ A | 0 | Null |
| ✓ B | ~~∞~~ 4 | ~~Null~~ A |
| ✓ C | ~~∞~~ 8 | ~~Null~~ B |
| ✓ D | ~~∞~~ 7 | ~~Null~~ C |
| E | ~~∞~~ ~~10~~ 9 | ~~Null~~ ~~F~~ D |
| ✓ F | ~~∞~~ 4 | ~~Null~~ C |
| ✓ G | ~~∞~~ ~~6~~ 2 | ~~Null~~ ~~I~~ F |
| ✓ H | ~~∞~~ ~~8~~ ~~7~~ 1 | ~~Null~~ ~~A~~ ~~I~~ G |
| ✓ I | ~~∞~~ 2 | ~~Null~~ C |

- Our next greedy choice is $CD$. So, I shall first update the neighbors of $D$ and I shall not consider $D$ again

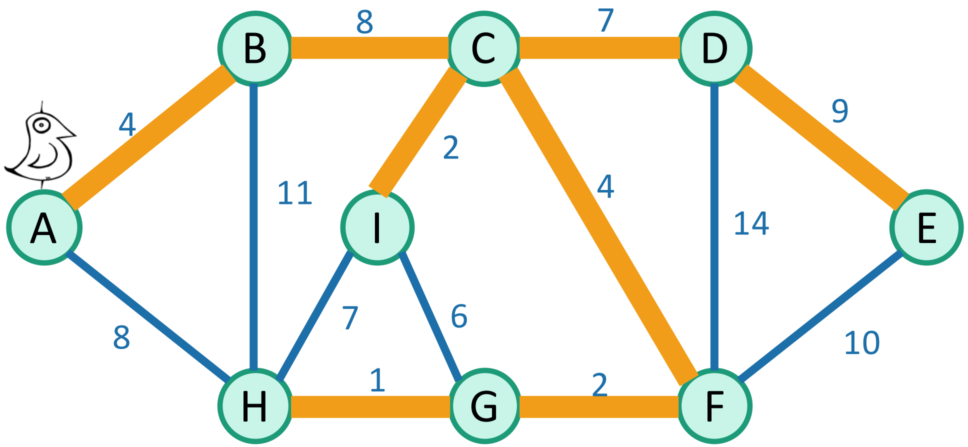
# How to find Minimum Spanning Trees




| | key | $\pi$ |
|---|---|---|
| A | 0 | Null |
| B | ~~∞~~ 4 | ~~Null~~ A |
| C | ~~∞~~ 8 | ~~Null~~ B |
| D | ~~∞~~ 7 | ~~Null~~ C |
| E | ~~∞~~ ~~10~~ 9 | ~~Null~~ ~~F~~ D |
| F | ~~∞~~ 4 | ~~Null~~ C |
| G | ~~∞~~ ~~6~~ 2 | ~~Null~~ ~~I~~ F |
| H | ~~∞~~ ~~8~~ ~~7~~ 1 | ~~Null~~ ~~A~~ ~~I~~ G |
| I | ~~∞~~ 2 | ~~Null~~ C |

- Our next greedy choice is $DE$. However, $E$'s neighbors are already updated
- Note choosing DE is not completing any cycle, but exploring its neighbors can
- Now the priority queue is empty, so we stop

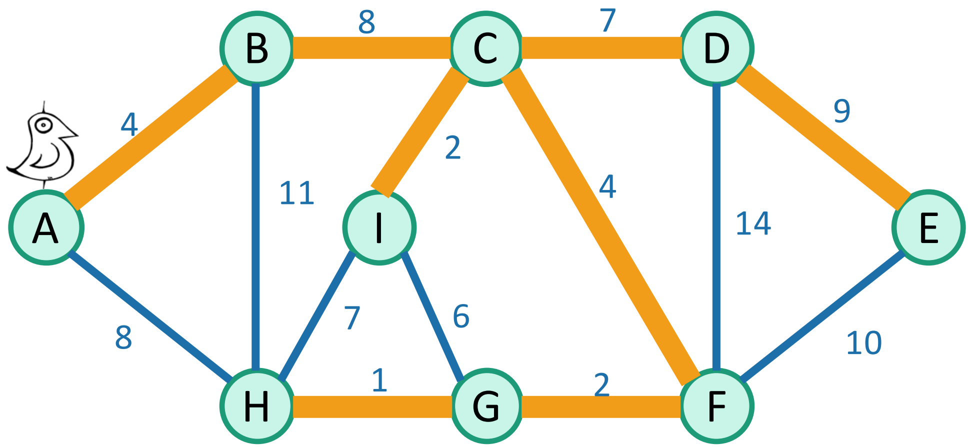
# How to find Minimum Spanning Trees




| | key | $\pi$ |
|---|---|---|
| A | 0 | Null |
| B | ~~∞~~ 4 | ~~Null~~ A |
| C | ~~∞~~ 8 | ~~Null~~ B |
| D | ~~∞~~ 7 | ~~Null~~ C |
| E | ~~∞~~ ~~10~~ 9 | ~~Null~~ ~~F~~ D |
| F | ~~∞~~ 4 | ~~Null~~ C |
| G | ~~∞~~ ~~6~~ 2 | ~~Null~~ ~~I~~ F |
| H | ~~∞~~ ~~8~~ ~~7~~ 1 | ~~Null~~ ~~A~~ ~~I~~ G |
| I | ~~∞~~ 2 | ~~Null~~ C |

- Our next greedy choice is $DE$. However, $E$'s neighbors are already updated
- Note choosing DE is not completing any cycle, but exploring its neighbors can
- Now the priority queue is empty, so we stop

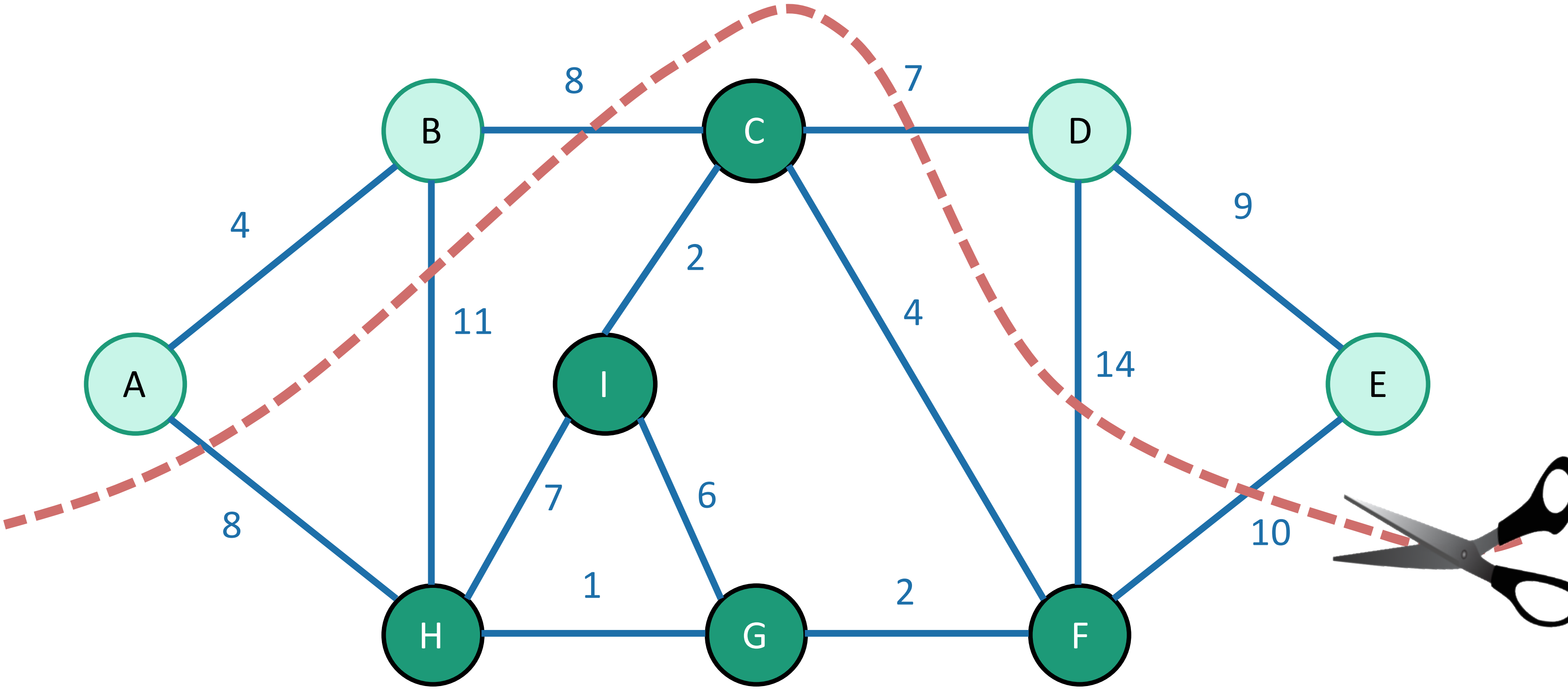
# How to find Minimum Spanning Trees




Prim($G$)
    select a source $s$
    **for** each vertex $u \in G.V$
        $u.key = \infty, u.\pi = Null$
    $s.key = 0$
    // Initialize a data structure for the vertices
    $Q = \phi$ // Priority queue
    **for** each vertex $u \in G.V$
        $Q.Insert(u)$
    **while** $Q \neq \phi$
        $u = Q.getMin()$
        **for** each neighbor $v \in G.Adj[u]$
            **if** v $\in Q$ $AND$ $w(u,v) < v.key$
                $v.key = w(u,v)$
                $v.\pi = u$

# Brief Aside

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

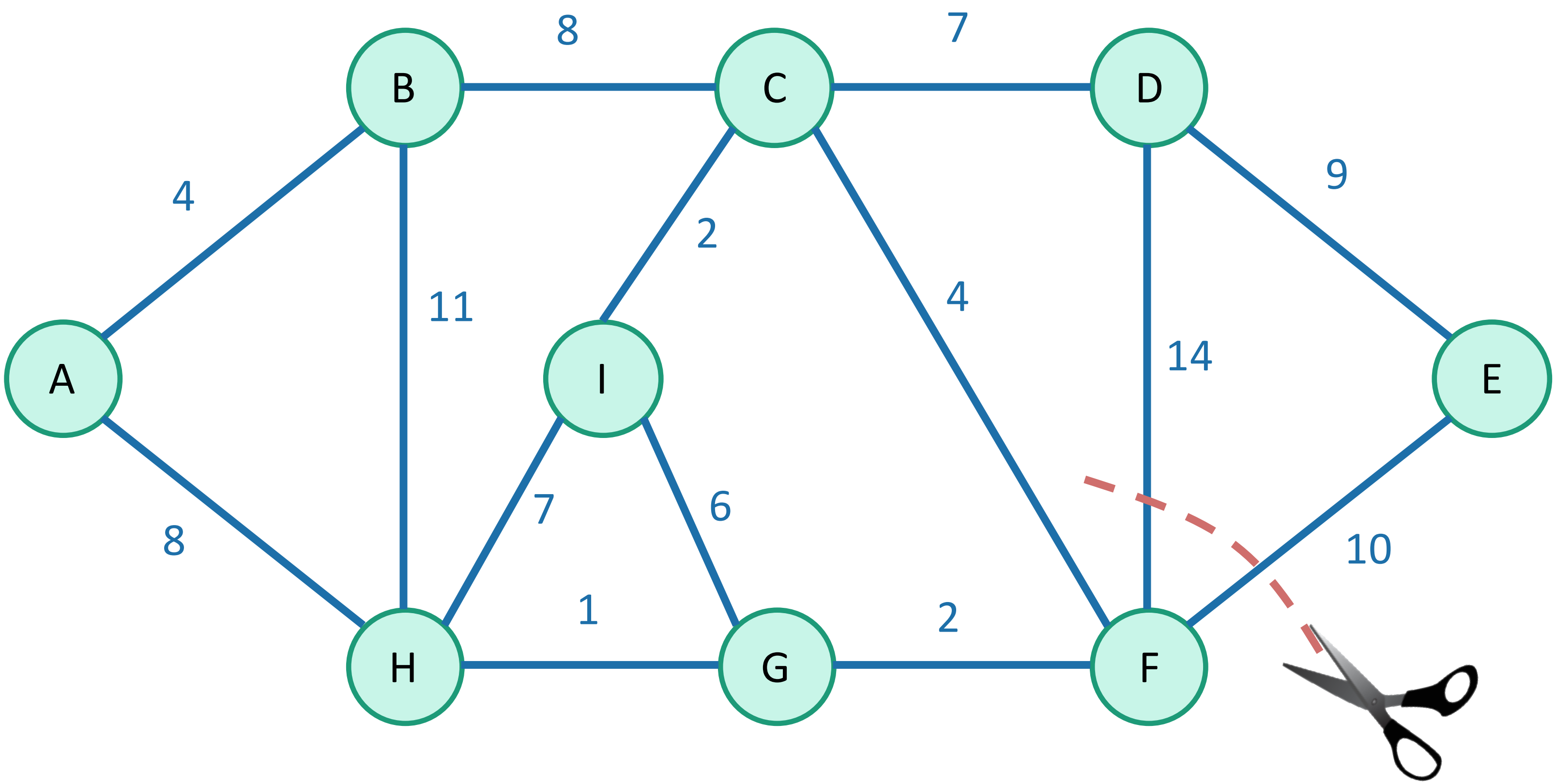



This is the cut **"{A,B,D,E} and {C,I,H,G,F}"**

*Source: Stanford, CS 161 course, Winter 2022*

# Cuts in Graphs

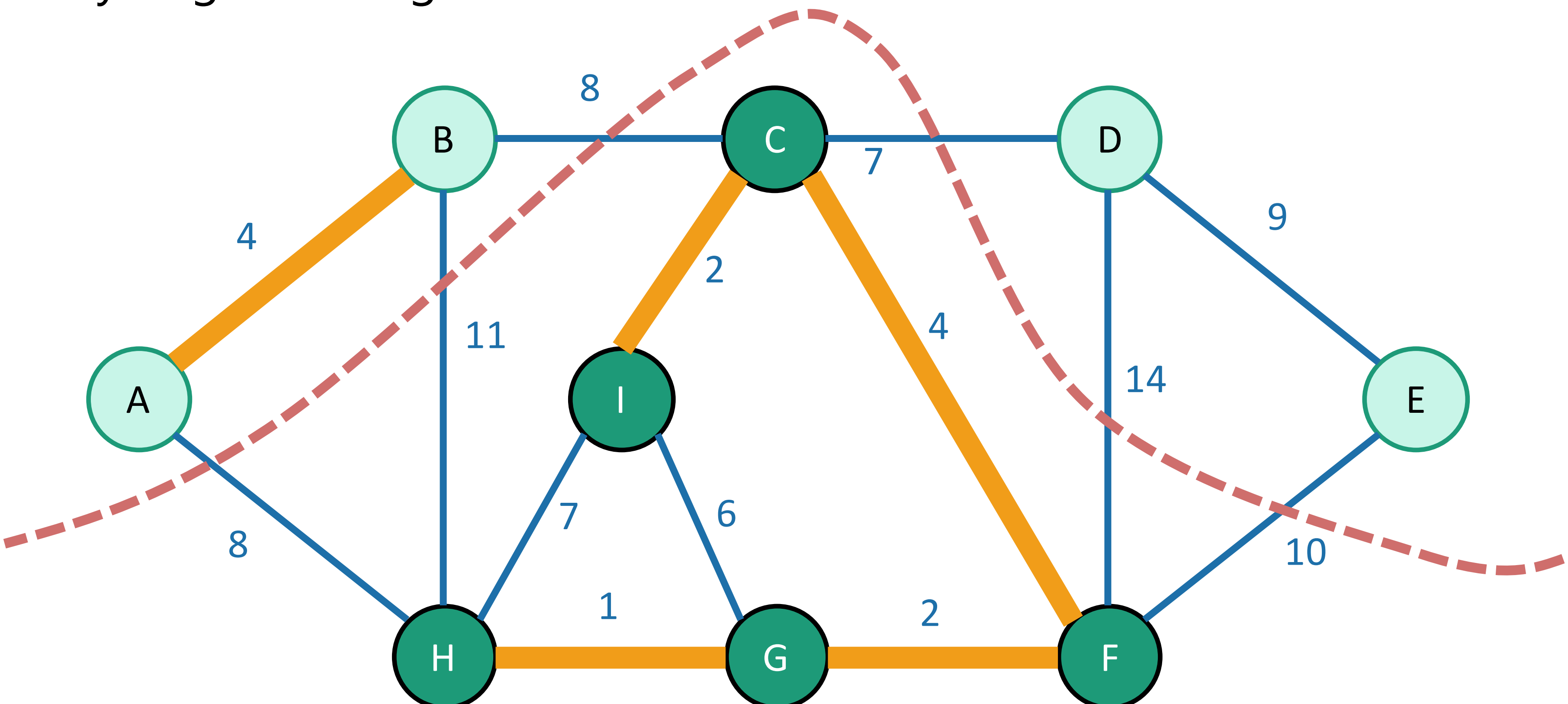
- This is **not** a cut. Cuts are partitions of vertices




Source: Stanford, CS 161 course, Winter 2022

# 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

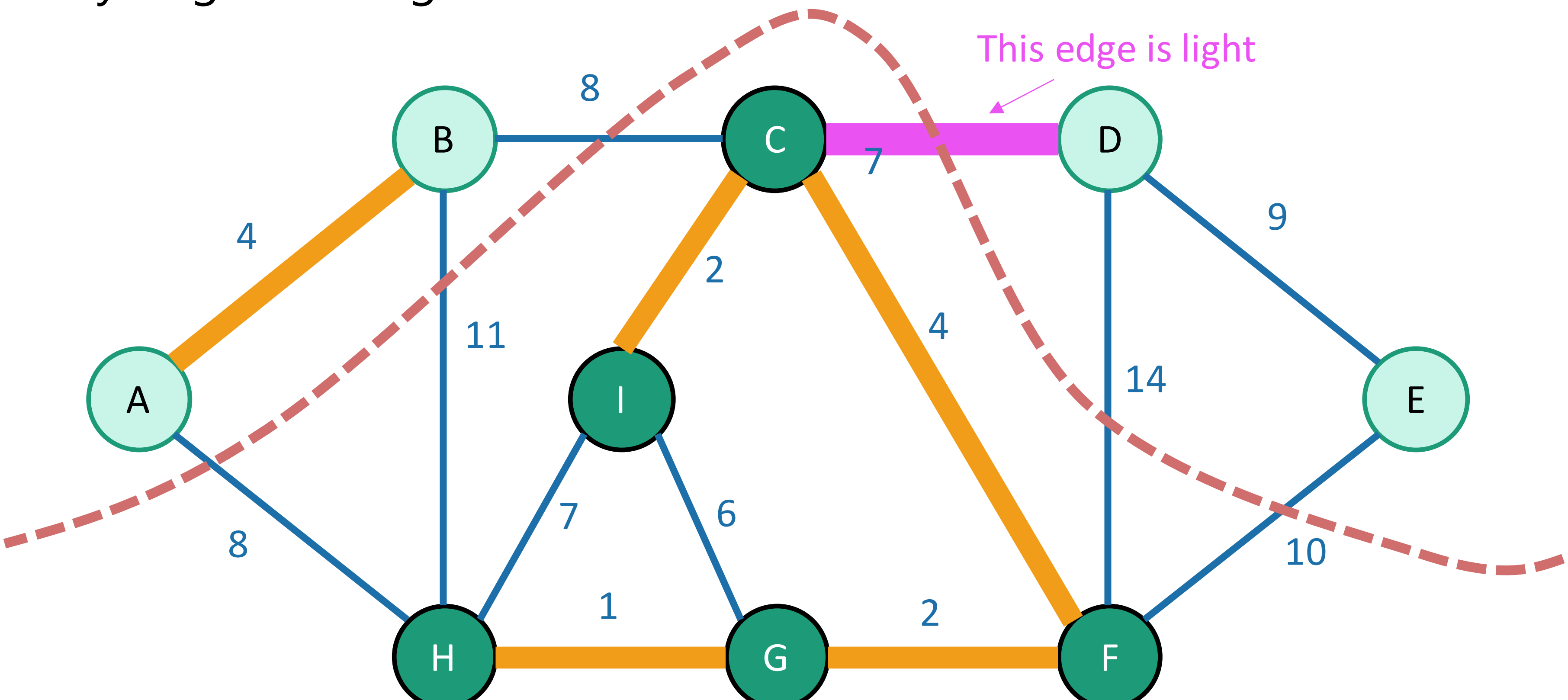



S is the set of **thick orange** edges

Source: Stanford, CS 161 course, Winter 2022

# 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

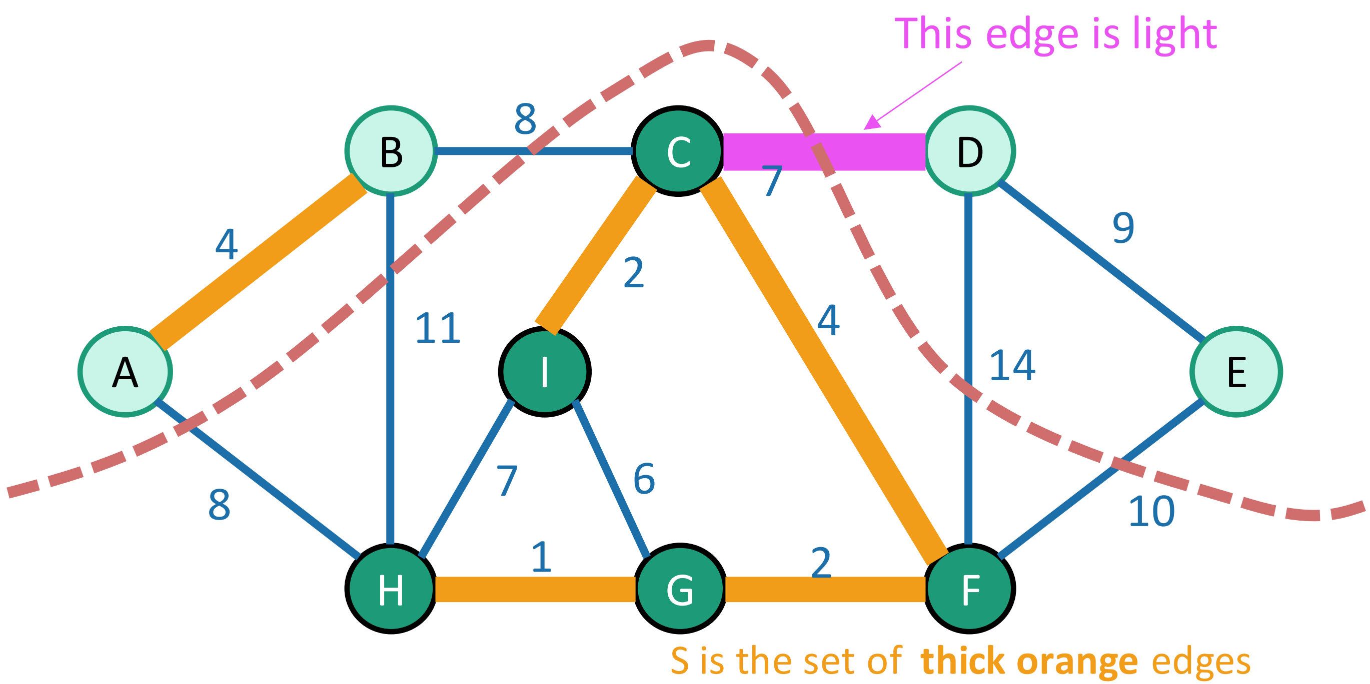



S is the set of **thick orange** edges

*Source: Stanford, CS 161 course, Winter 2022*

# 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}}

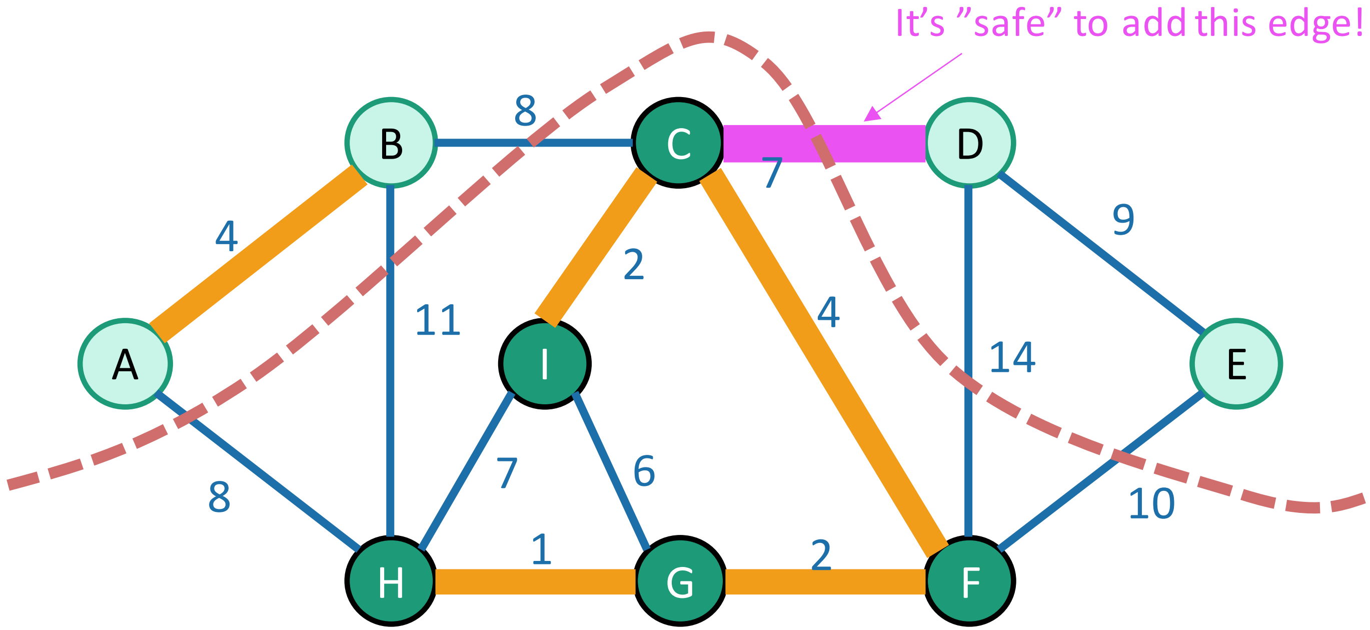



Source: Stanford, CS 161 course, Winter 2022

# 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}}

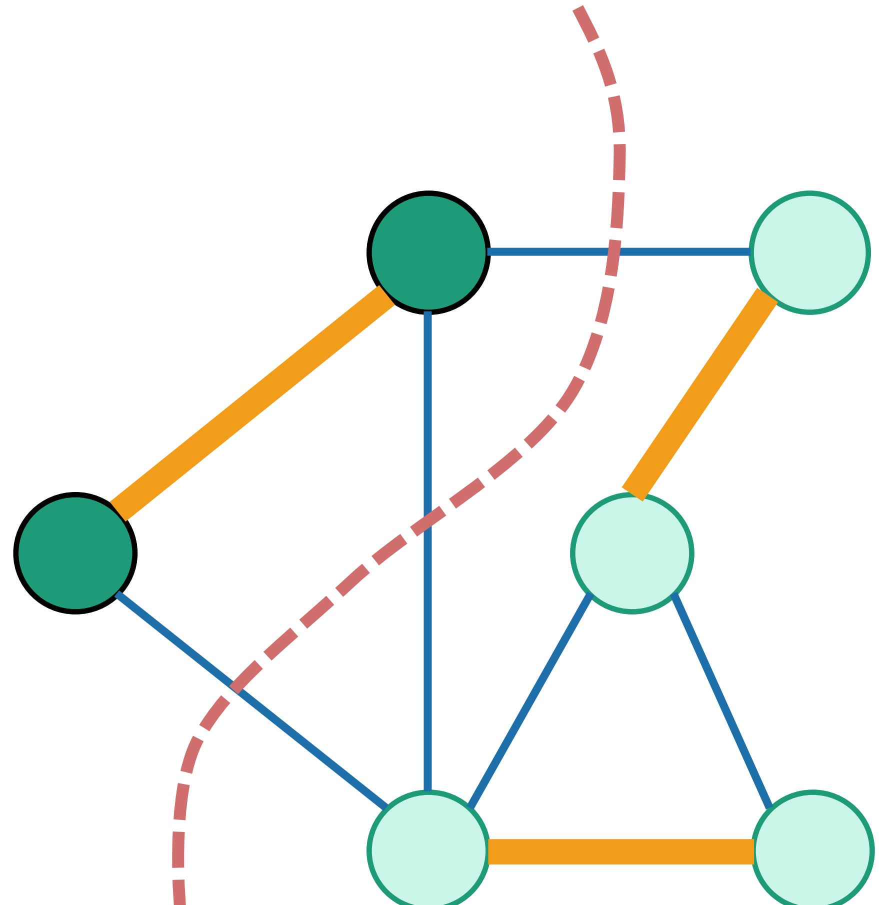
**If we haven't ruled out the possibility of success so far, then adding a light edge still won't rule it out.**




Source: Stanford, CS 161 course, Winter 2022

# Proof of Lemma

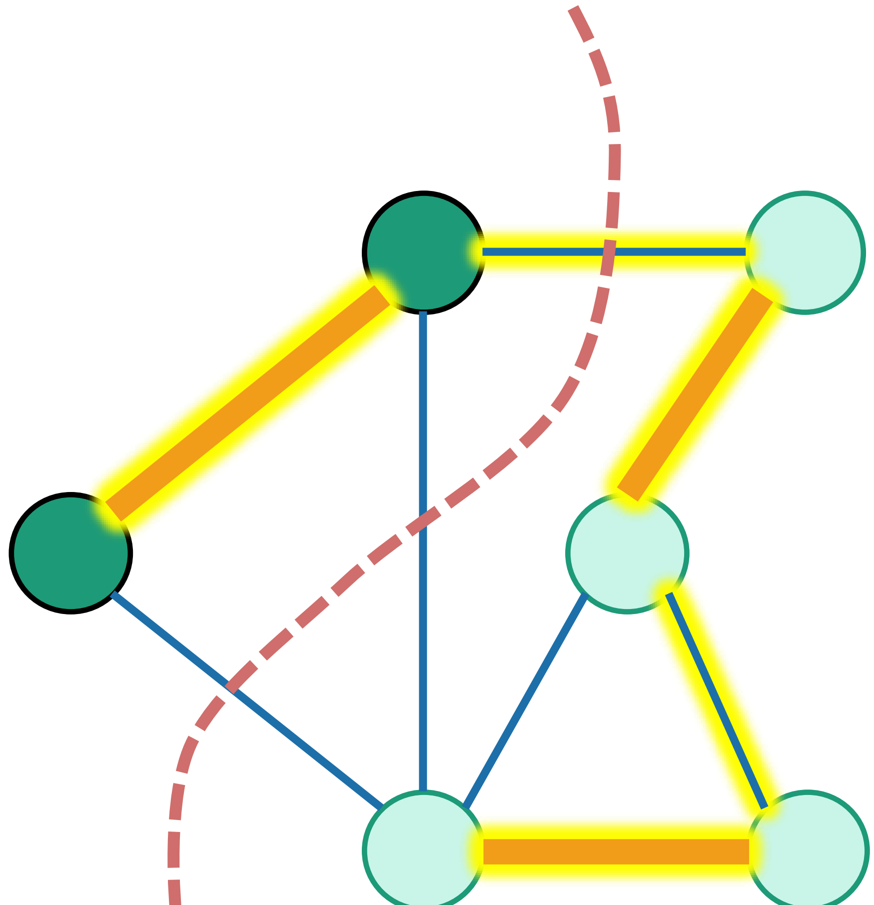
- Assume that we have:
  - a **cut** that respects S



Source: Stanford, CS 161 course, Winter 2022

# Proof of Lemma

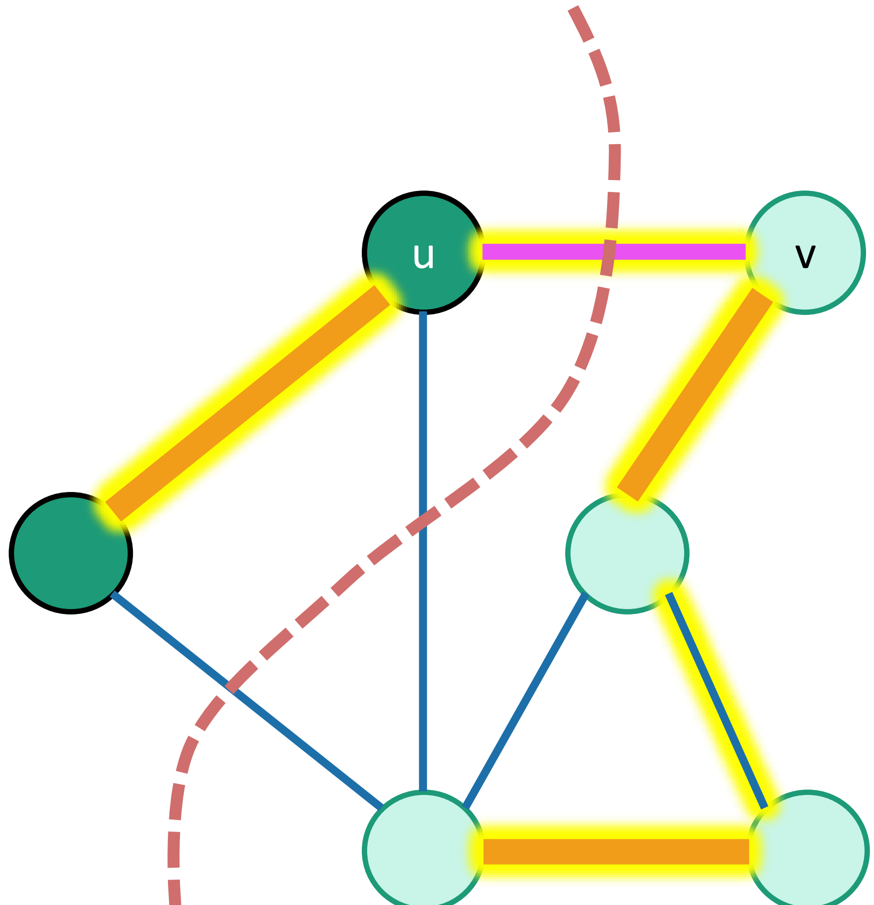
- Assume that we have:
  - a **cut** that respects S
  - S is part of some MST T
- Say that **{u,v}** is light
  - lowest cost crossing the cut



*Source: Stanford, CS 161 course, Winter 2022*

# 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.

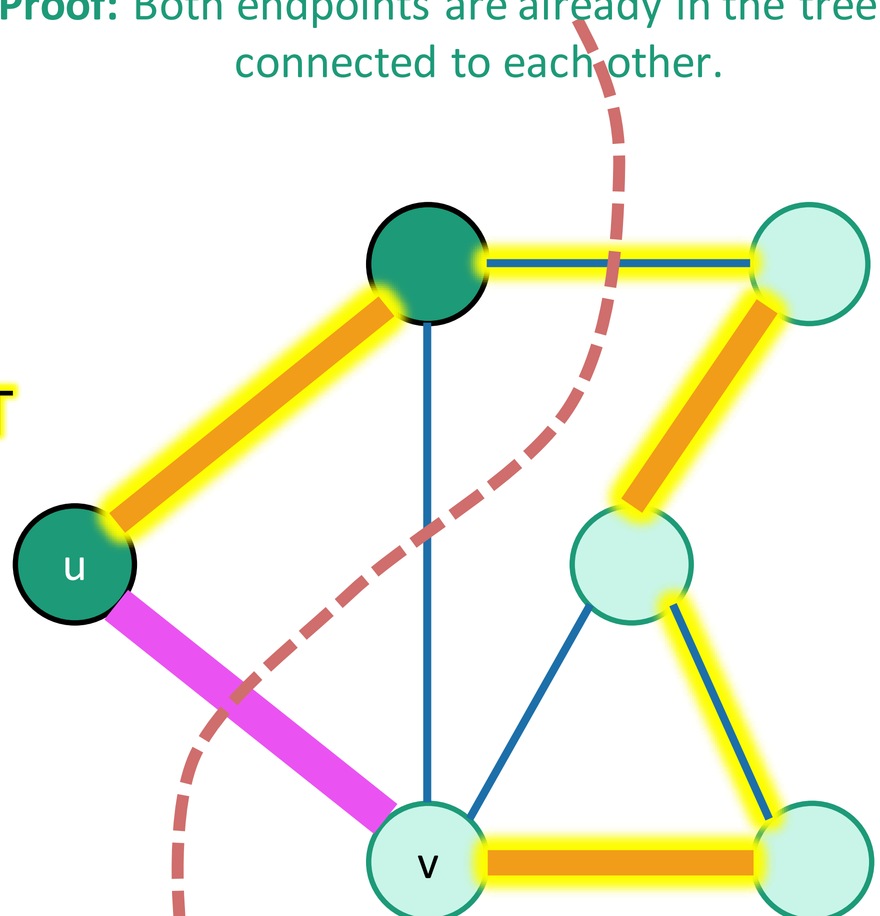



*Source: Stanford, CS 161 course, Winter 2022*

# 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

**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.



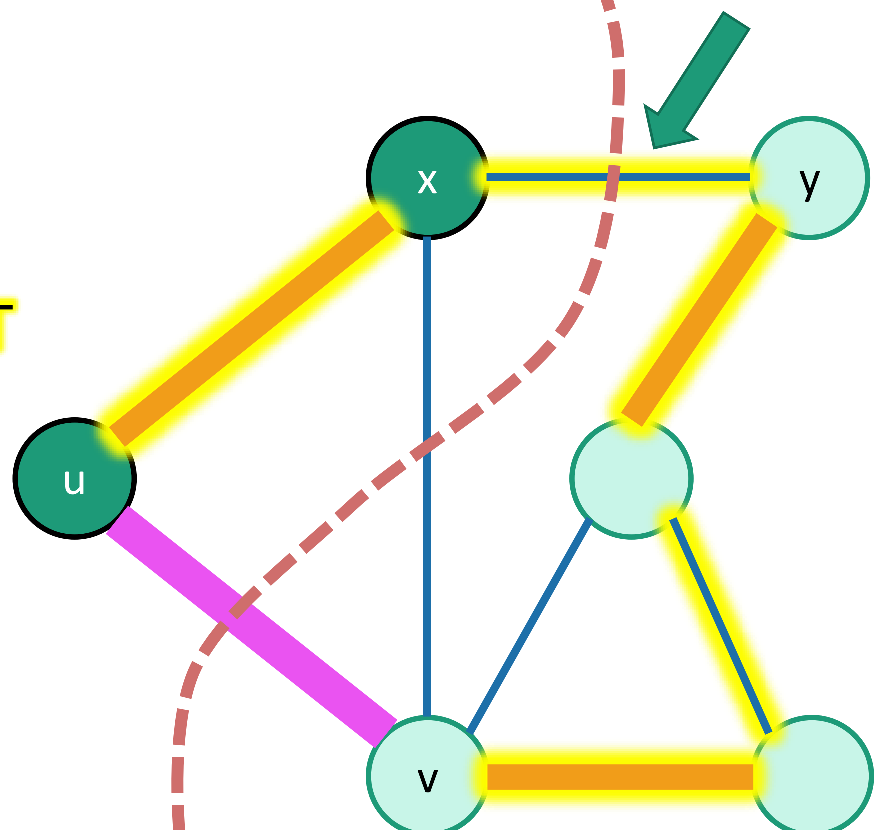

*Source: Stanford, CS 161 course, Winter 2022*

# 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

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

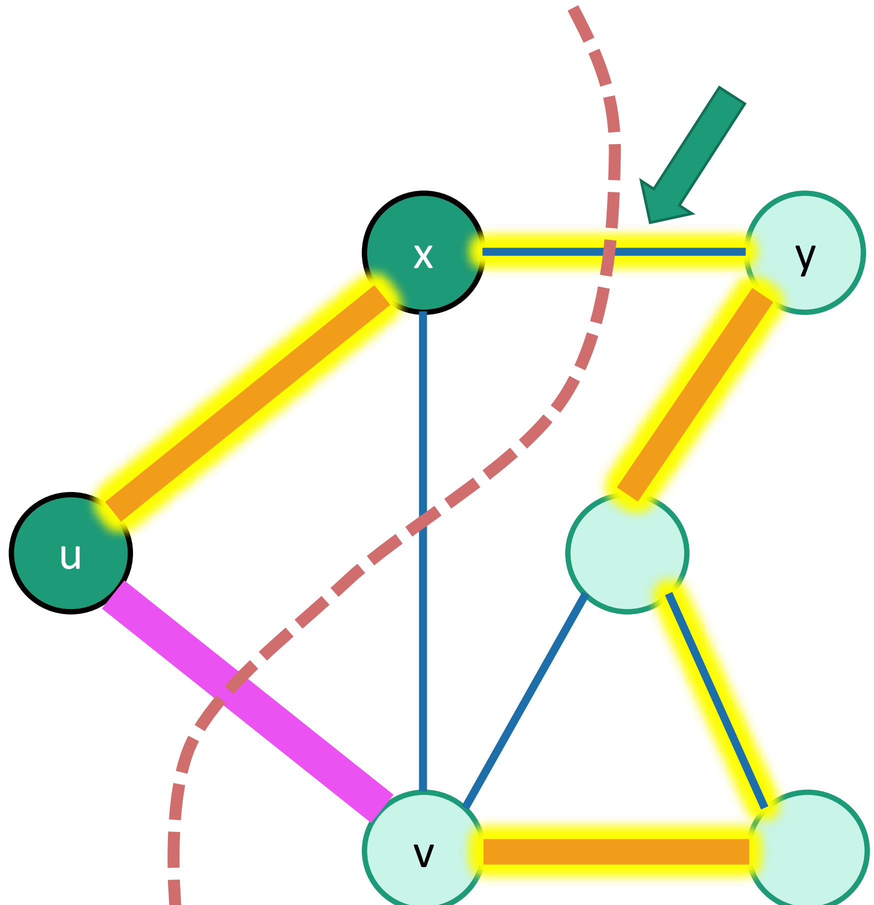
**Proof:** Both endpoints are already in the tree and connected to each other.




*Source: Stanford, CS 161 course, Winter 2022*

# Proof of Lemma

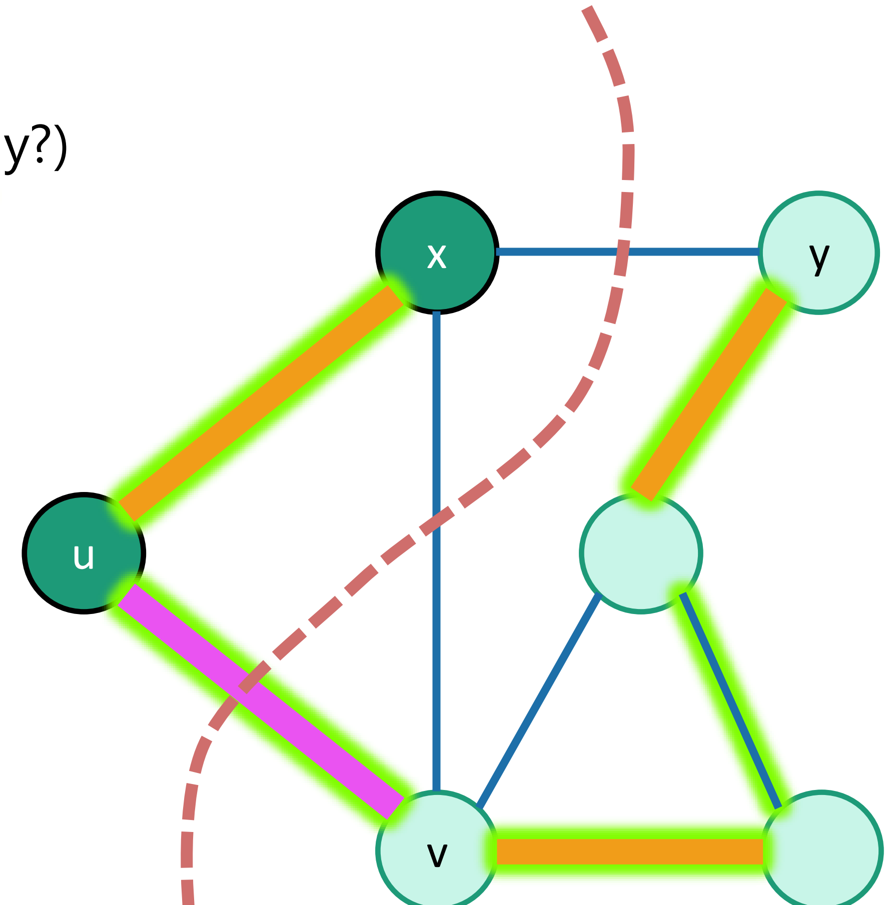
- Consider swapping {u,v} for {x,y} in T
  - Call the resulting tree T'




Source: Stanford, CS 161 course, Winter 2022

# Proof of Lemma

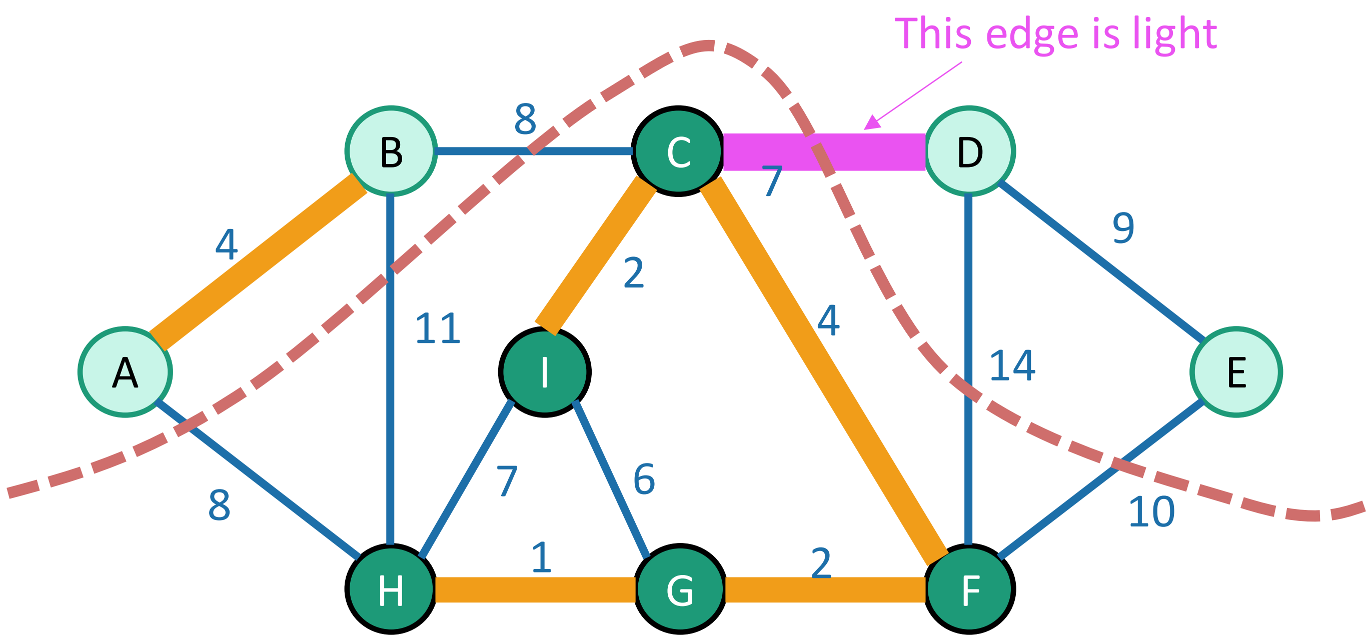
- Consider swapping {u,v} for {x,y} in T
  - Call the resulting tree T'
- Claim: T' is still an MST
  - It is still a spanning tree (why?)
  - It has cost at most that of T
    - Because {u,v} was light
  - T had minimal cost
  - So T' does too
- So T' is an MST containing S and {u,v}
  - This is what we wanted




*Source: Stanford, CS 161 course, Winter 2022*

# 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}}



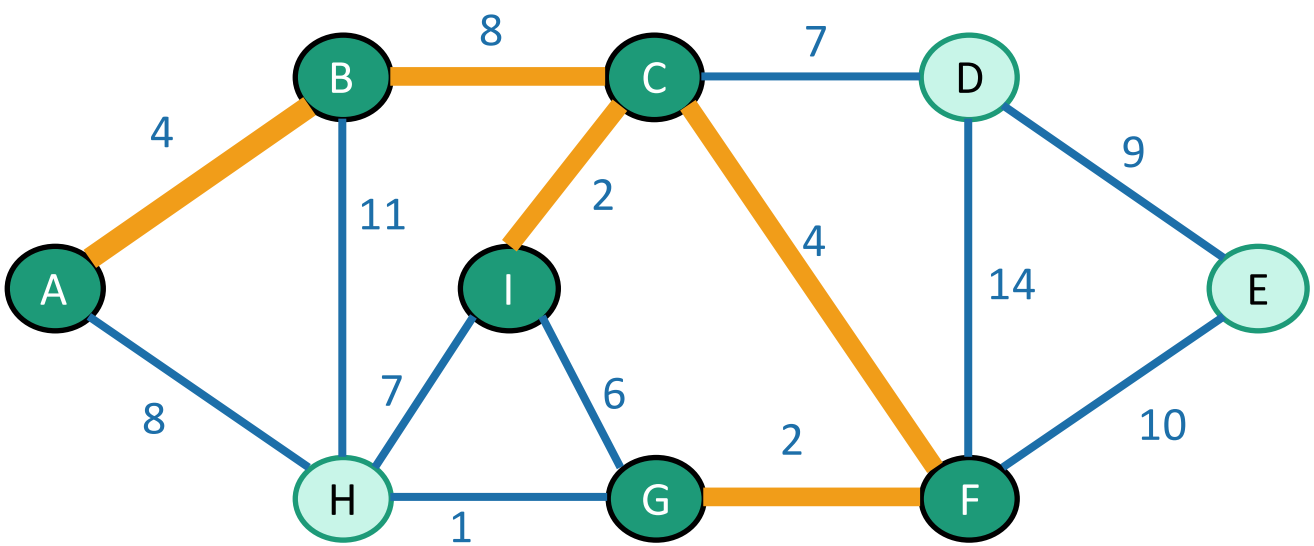

S is the set of **thick orange** edges

*Source: Stanford, CS 161 course, Winter 2022*

# 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.

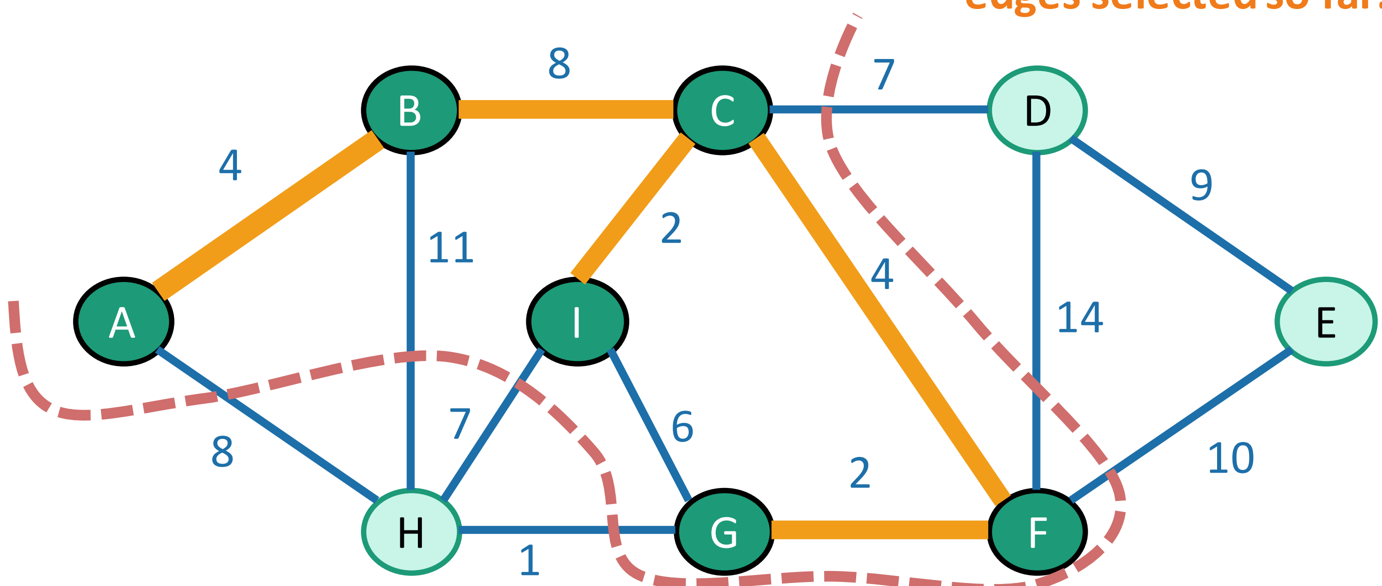



Source: Stanford, CS 161 course, Winter 2022

# 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.

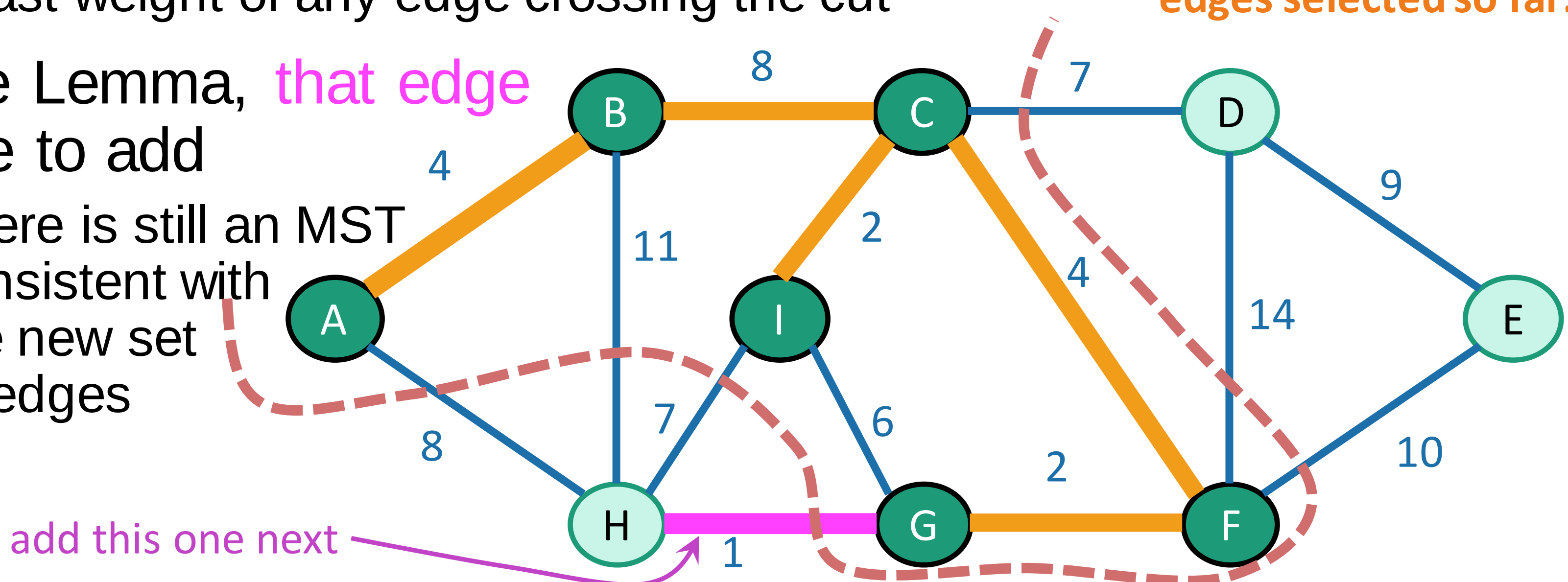



Source: Stanford, CS 161 course, Winter 2022

# 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
- 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.




Source: Stanford, CS 161 course, Winter 2022

# Partway through Prim

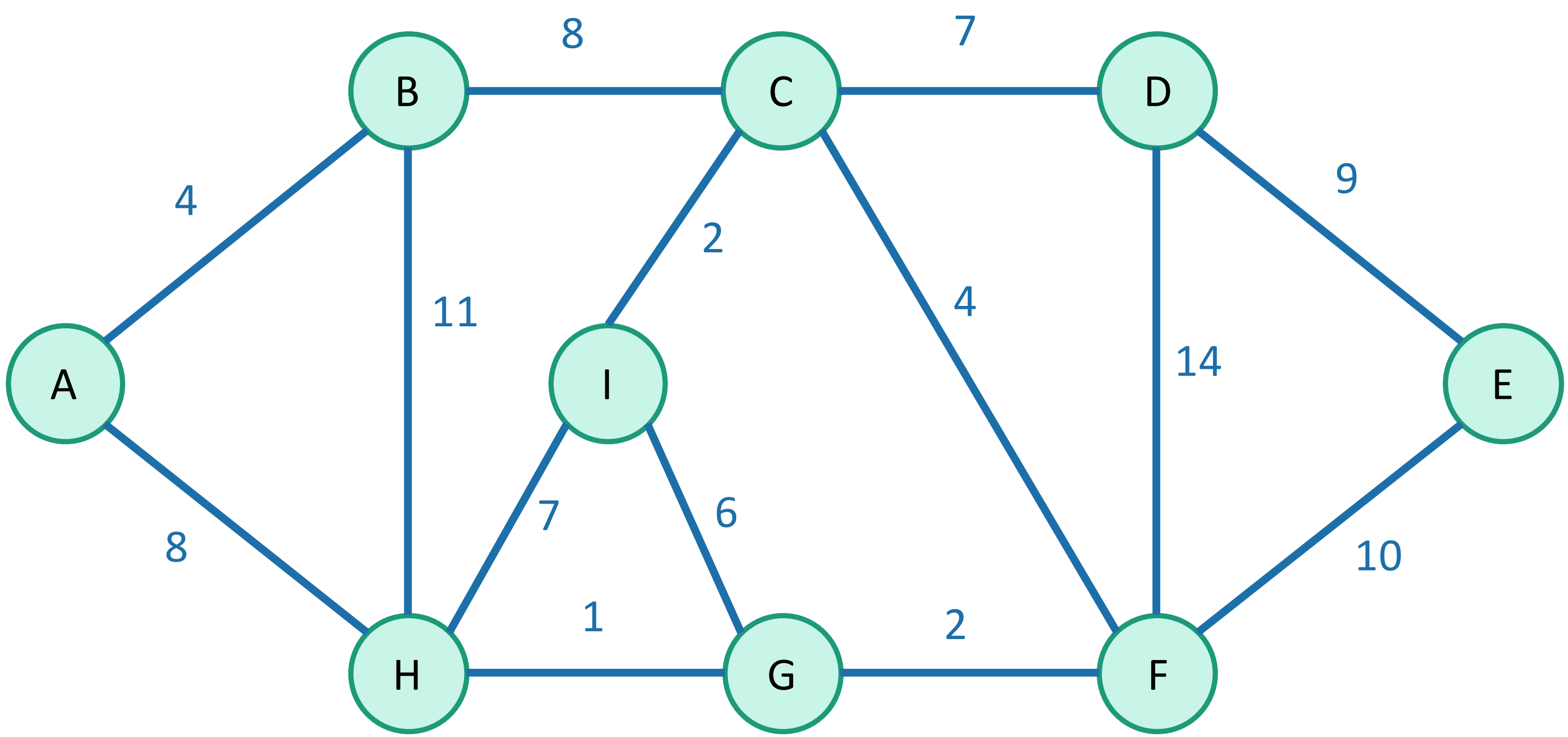
- Our greedy choices **don't rule out success**.

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

*Source: Stanford, CS 161 course, Winter 2022*

# That's not the only greedy algorithm for MST!

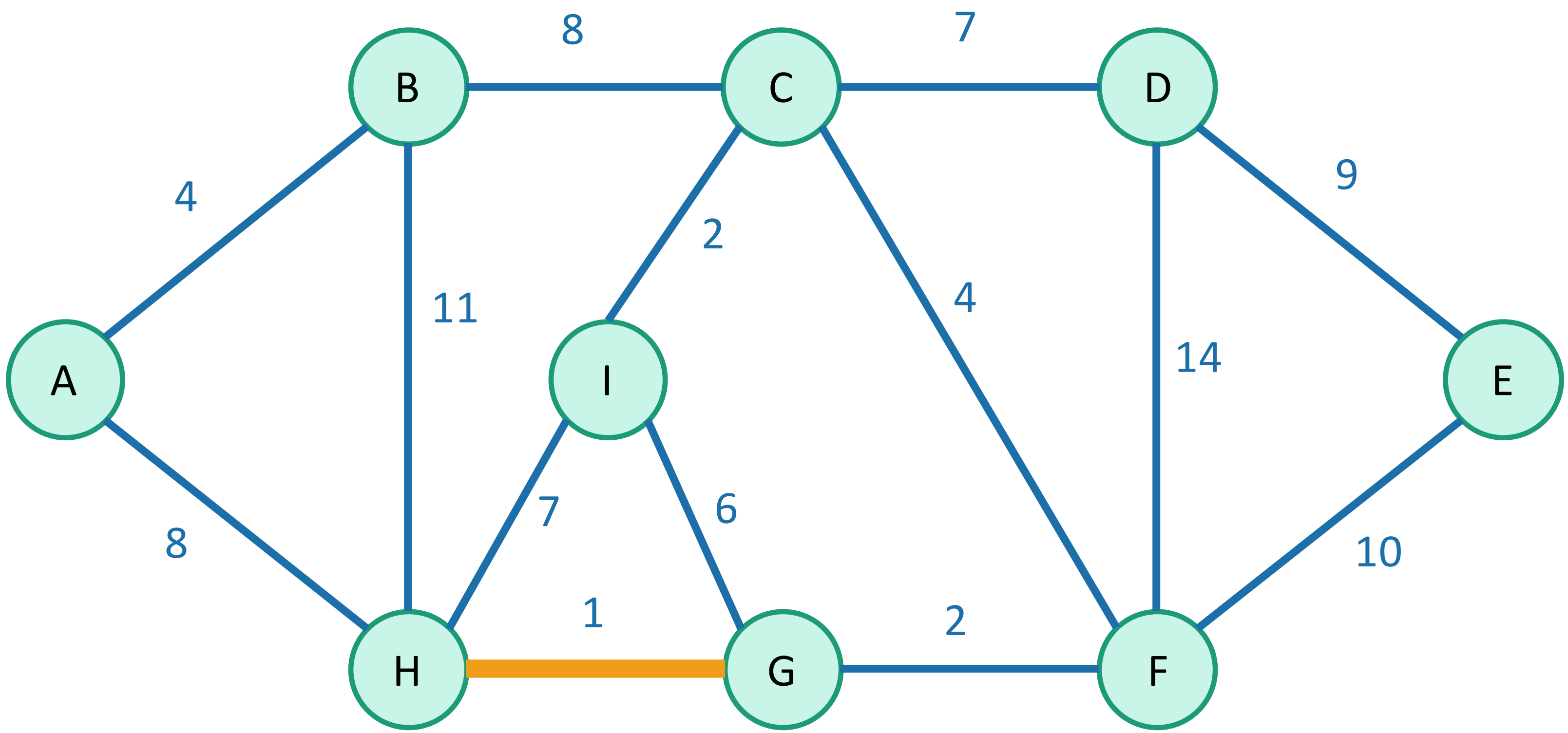
- what if we just always take the cheapest edge?
- whether or not it's connected to what we have so far?




Source: Stanford, CS 161 course, Winter 2022

# That's not the only greedy algorithm for MST!

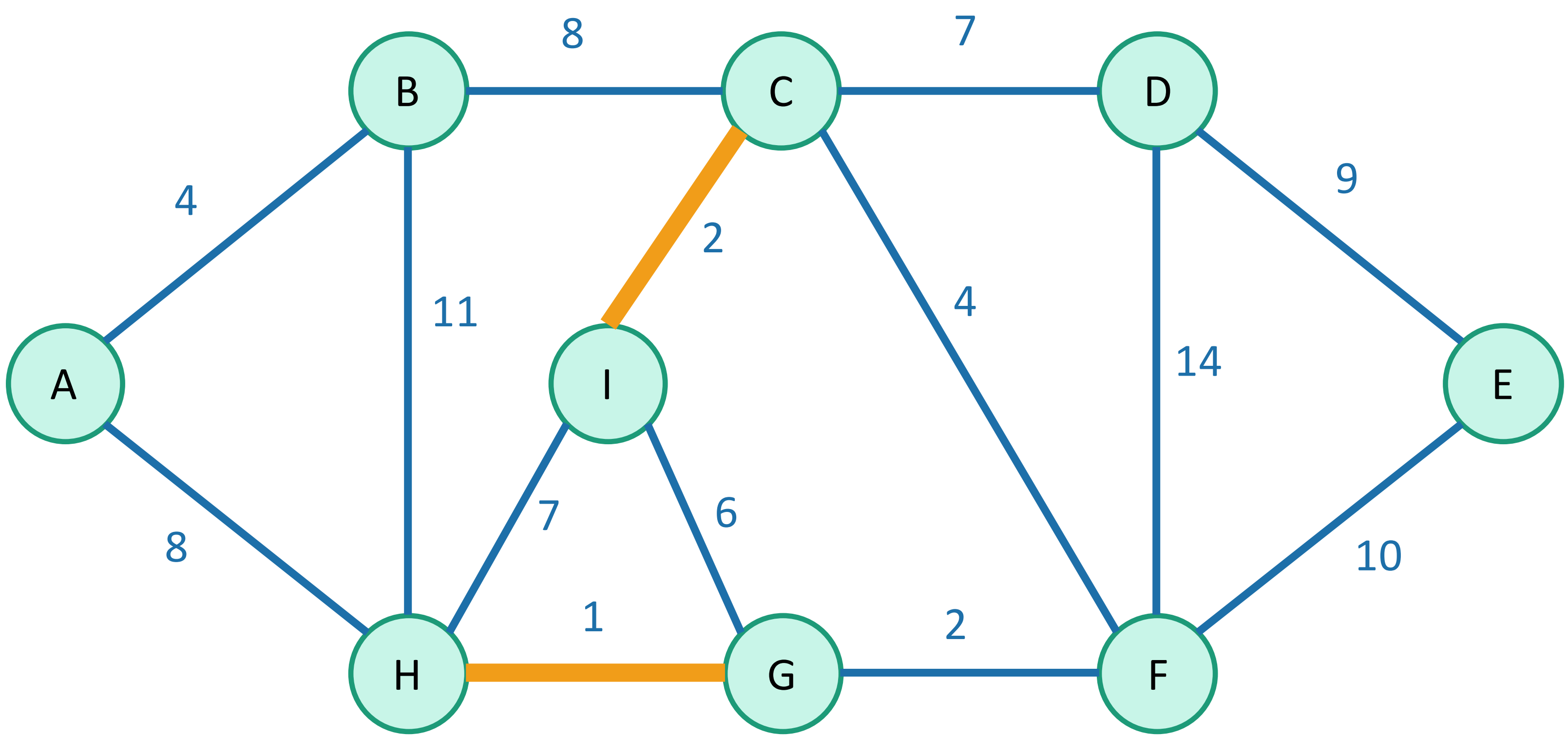
- what if we just always take the cheapest edge?
- whether or not it's connected to what we have so far?




Source: Stanford, CS 161 course, Winter 2022

# That's not the only greedy algorithm for MST!

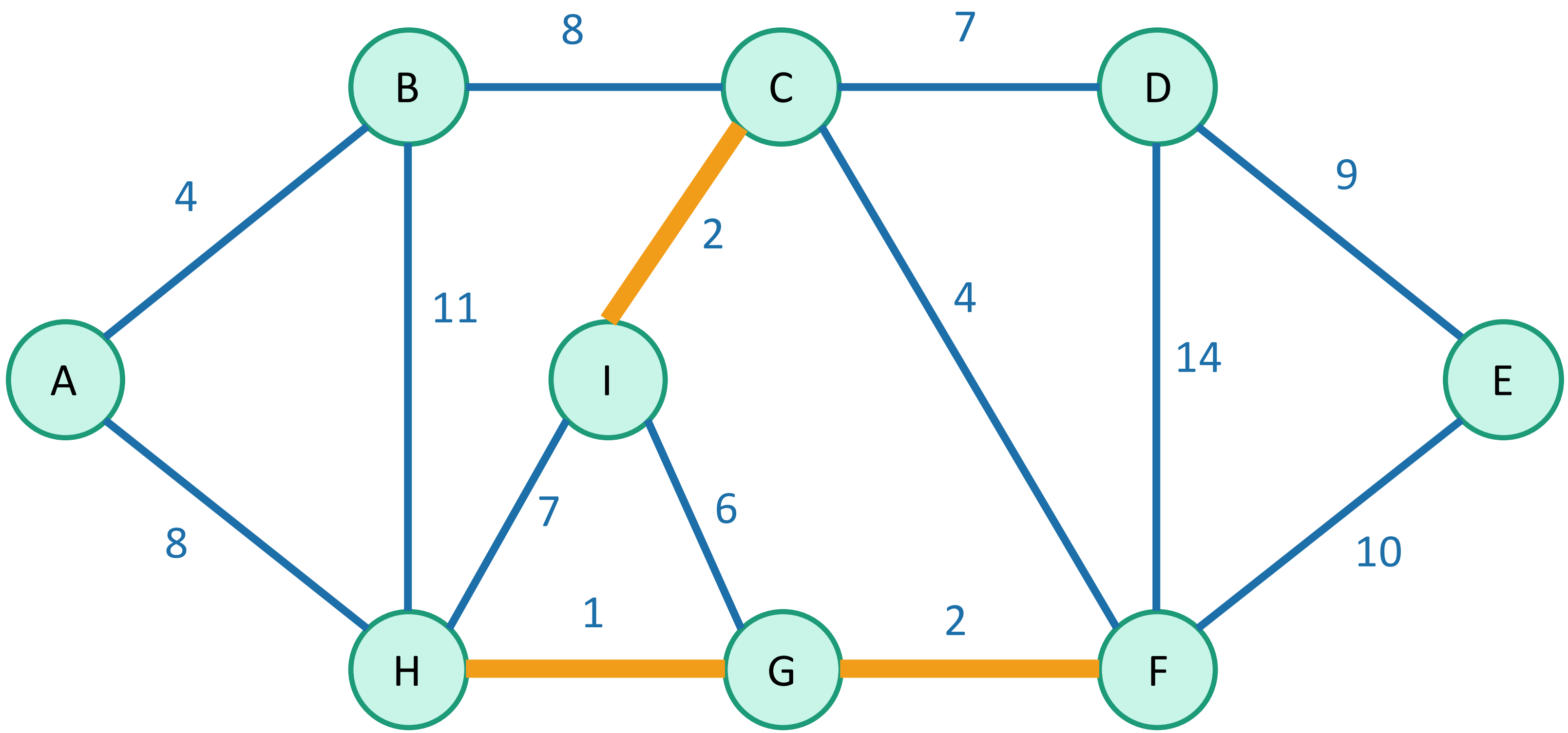
- what if we just always take the cheapest edge?
- whether or not it's connected to what we have so far?




Source: Stanford, CS 161 course, Winter 2022

# That's not the only greedy algorithm for MST!

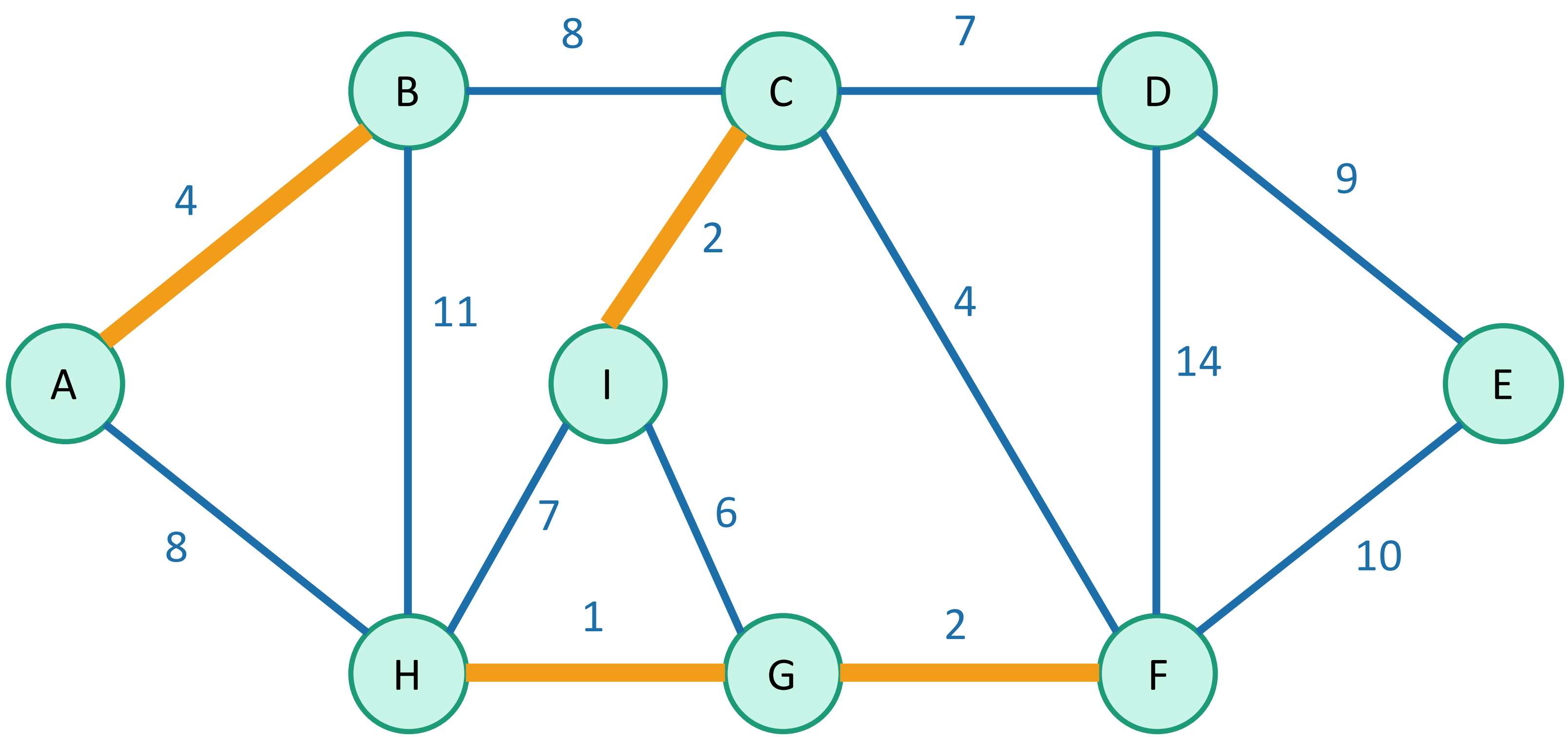
- what if we just always take the cheapest edge?
- whether or not it's connected to what we have so far?




Source: Stanford, CS 161 course, Winter 2022

# That's not the only greedy algorithm for MST!

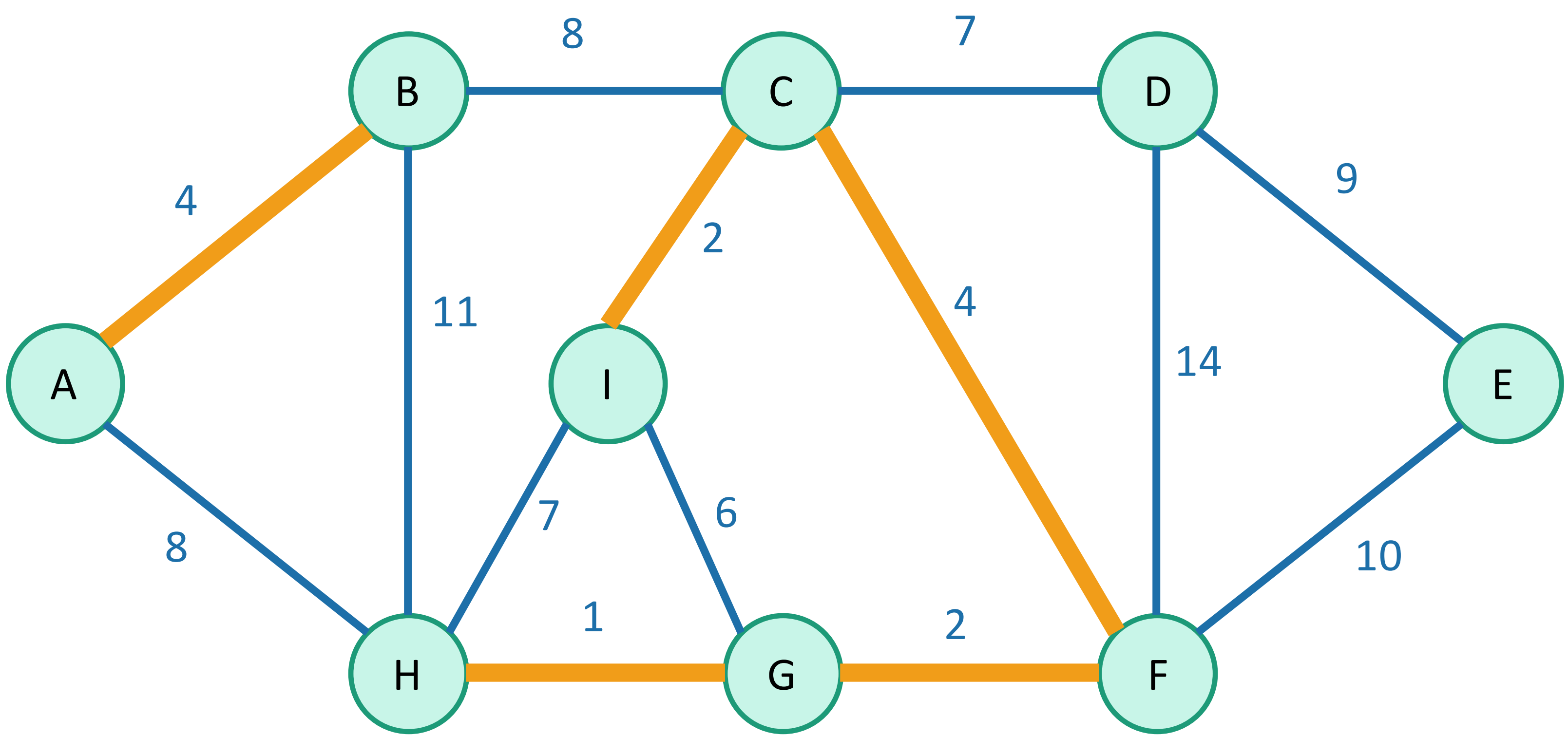
- what if we just always take the cheapest edge?
- whether or not it's connected to what we have so far?




Source: Stanford, CS 161 course, Winter 2022

# That's not the only greedy algorithm for MST!

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

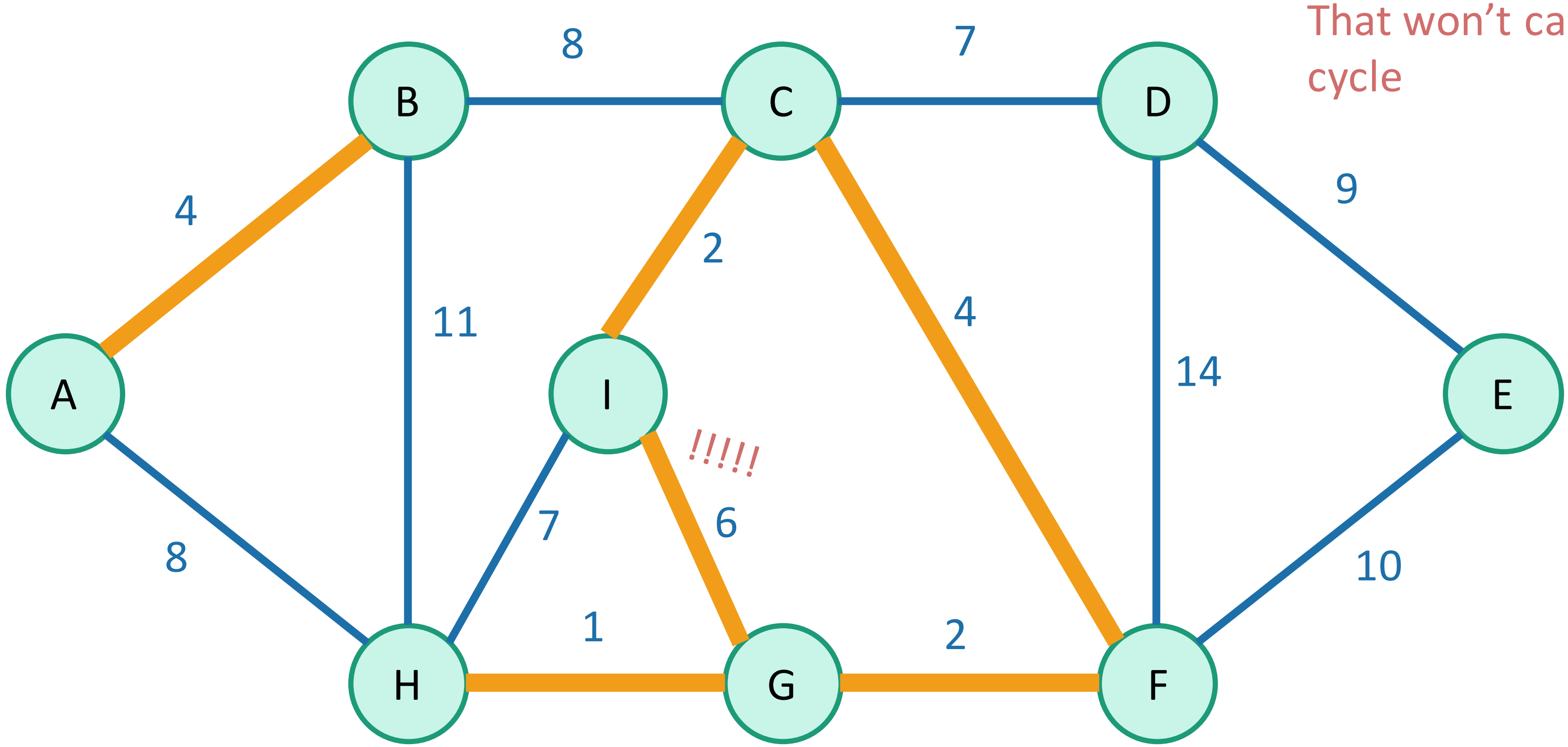



*Source: Stanford, CS 161 course, Winter 2022*

# That's not the only greedy algorithm for MST!

- 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

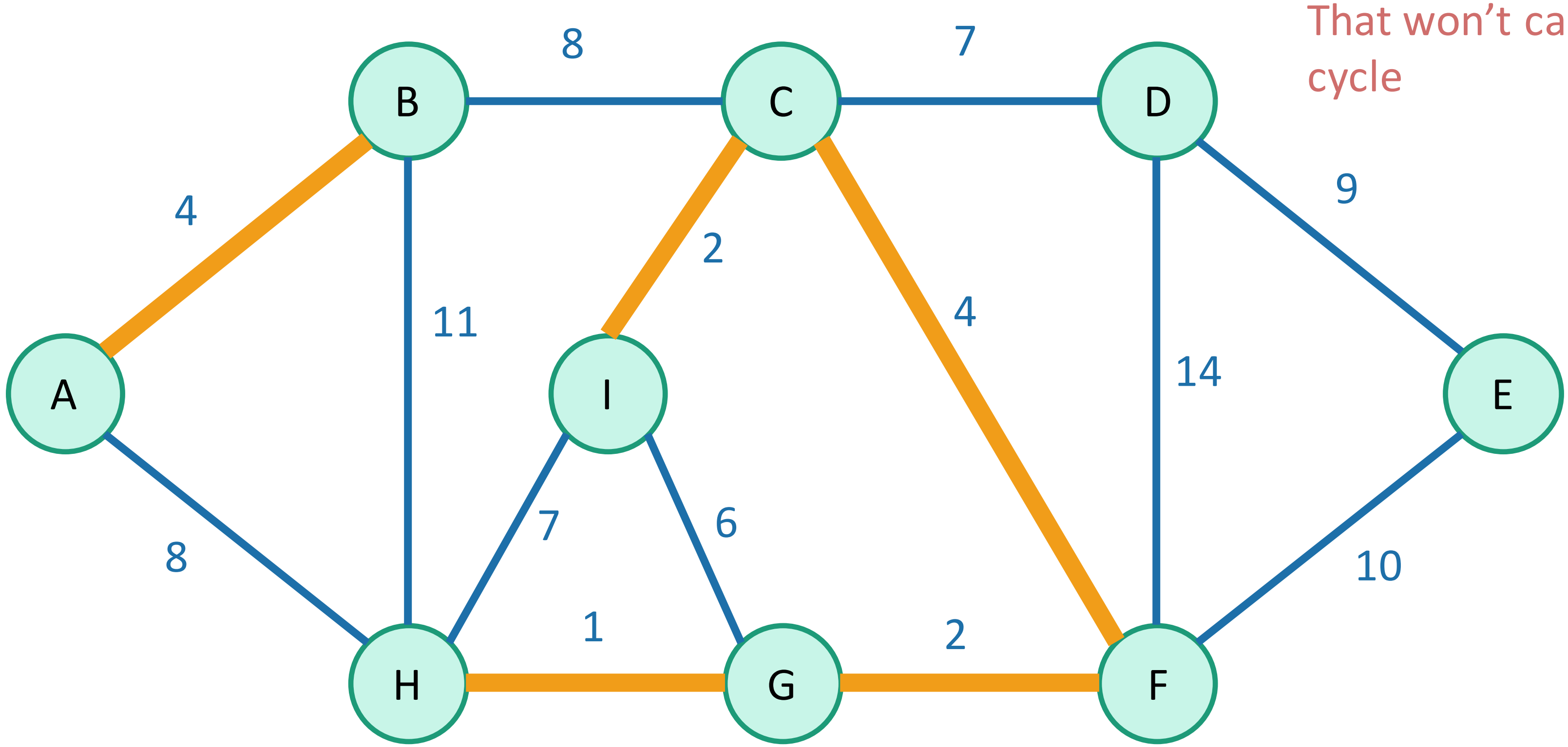



*Source: Stanford, CS 161 course, Winter 2022*

# That's not the only greedy algorithm for MST!

- 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

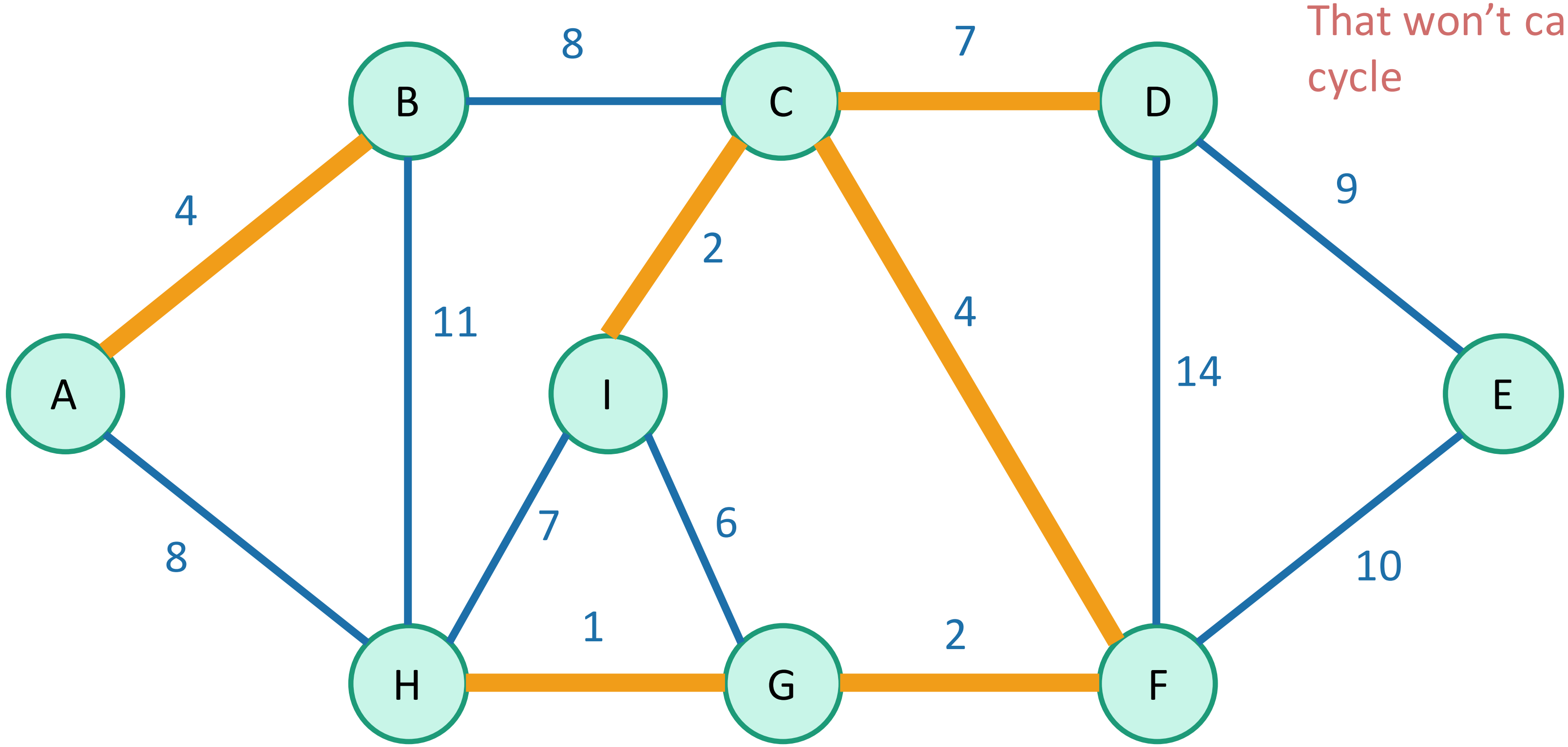



Source: Stanford, CS 161 course, Winter 2022

# That's not the only greedy algorithm for MST!

- 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

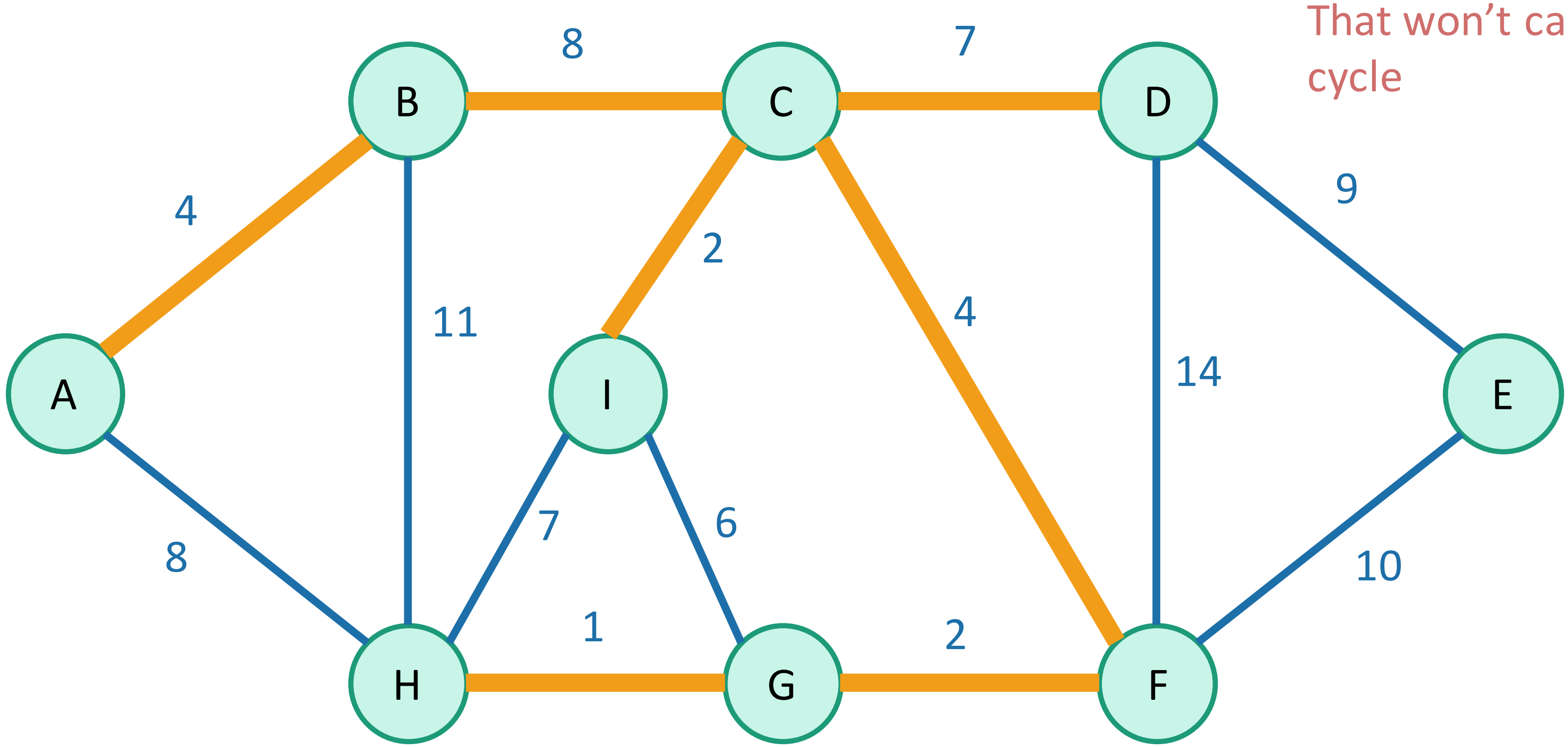



Source: Stanford, CS 161 course, Winter 2022

# That's not the only greedy algorithm for MST!

- 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

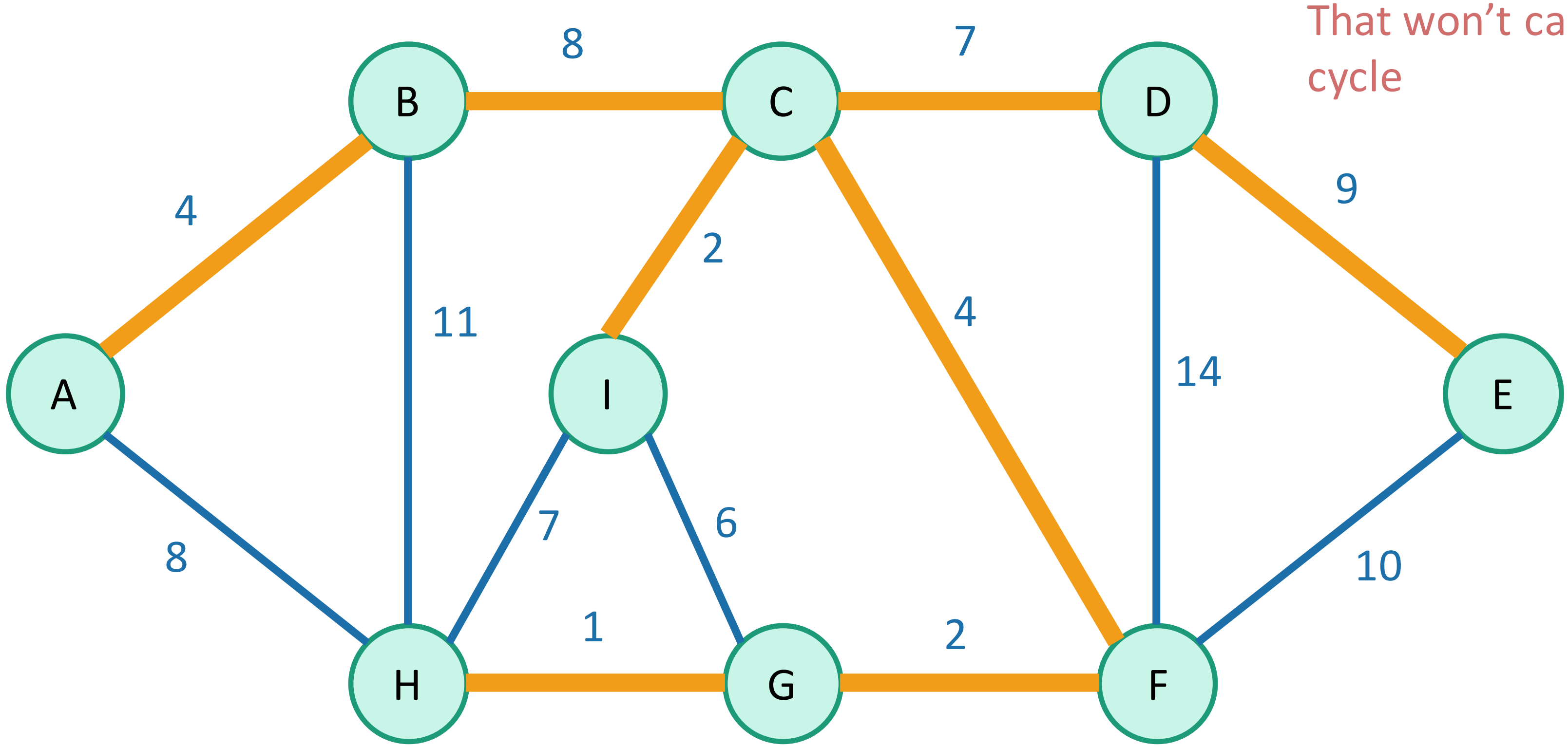



Source: Stanford, CS 161 course, Winter 2022

# That's not the only greedy algorithm for MST!

- 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




Source: Stanford, CS 161 course, Winter 2022

# We’ve Reached 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

$|E|$ 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 $|E|$ iterations of the for loop:
  - Check if adding e would cause a cycle…

*Source: Stanford, CS 161 course, Winter 2022*

# Two questions

- Does it work?
  - That is, does it actually return a MST?
- How do we actually implement this?
  - The pseudocode above says "slowKruskal" …

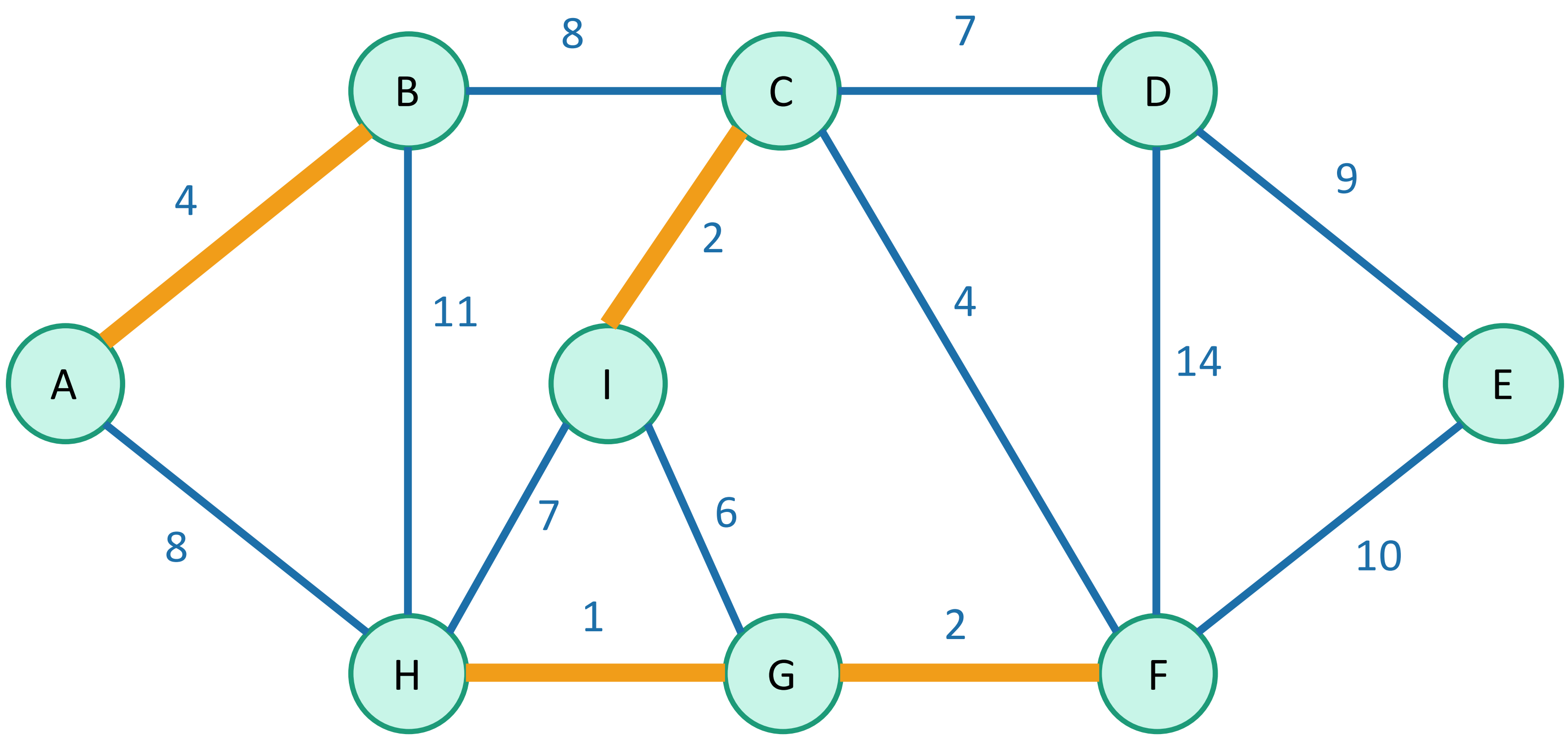
Let's do this one first

Source: Stanford, CS 161 course, Winter 2022

# At each step of Kruskal's

- We are maintaining a forest

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

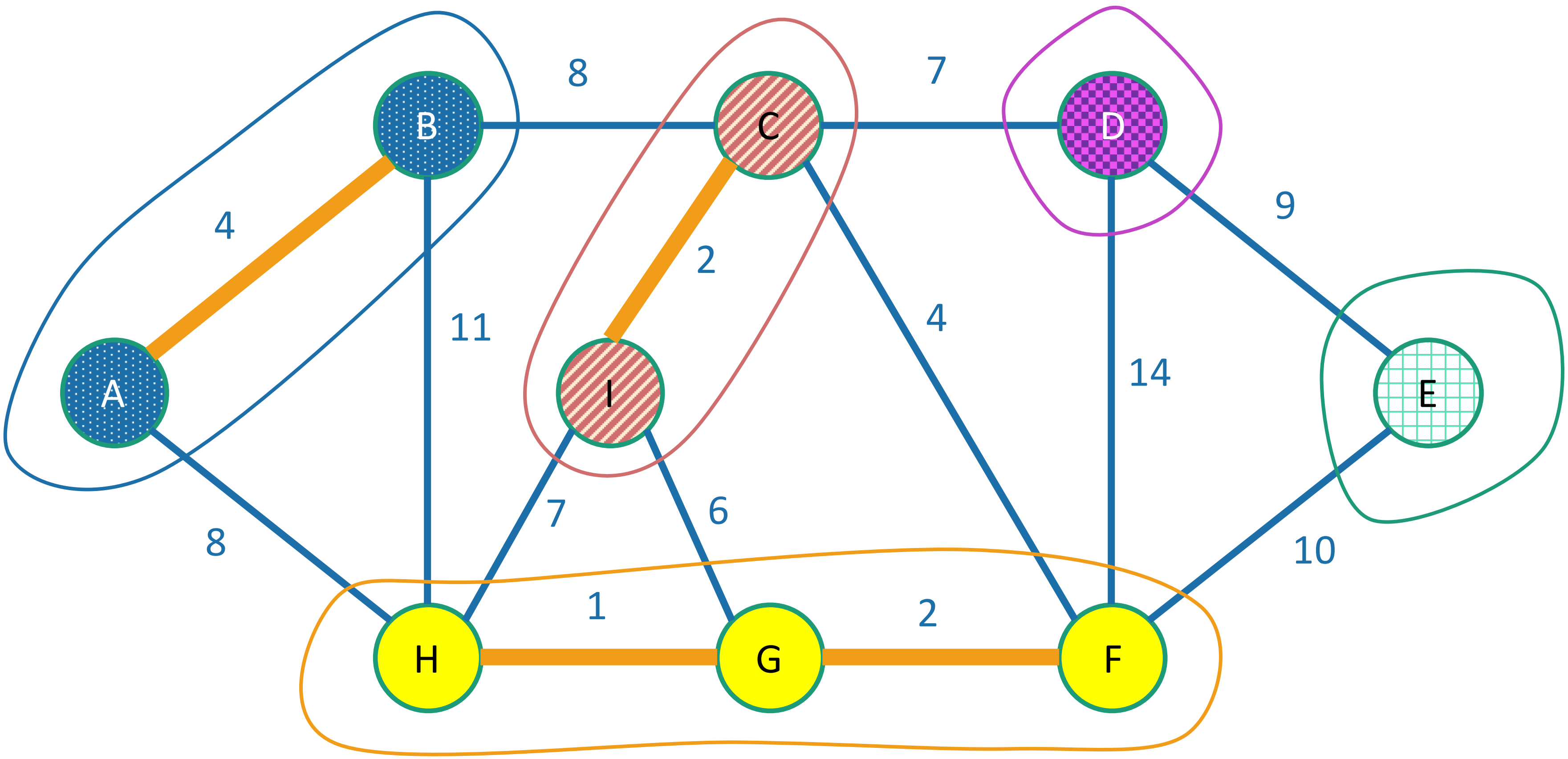



*Source: Stanford, CS 161 course, Winter 2022*

# At each step of Kruskal's

- We are maintaining a forest
- When we add an edge, we merge two trees

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

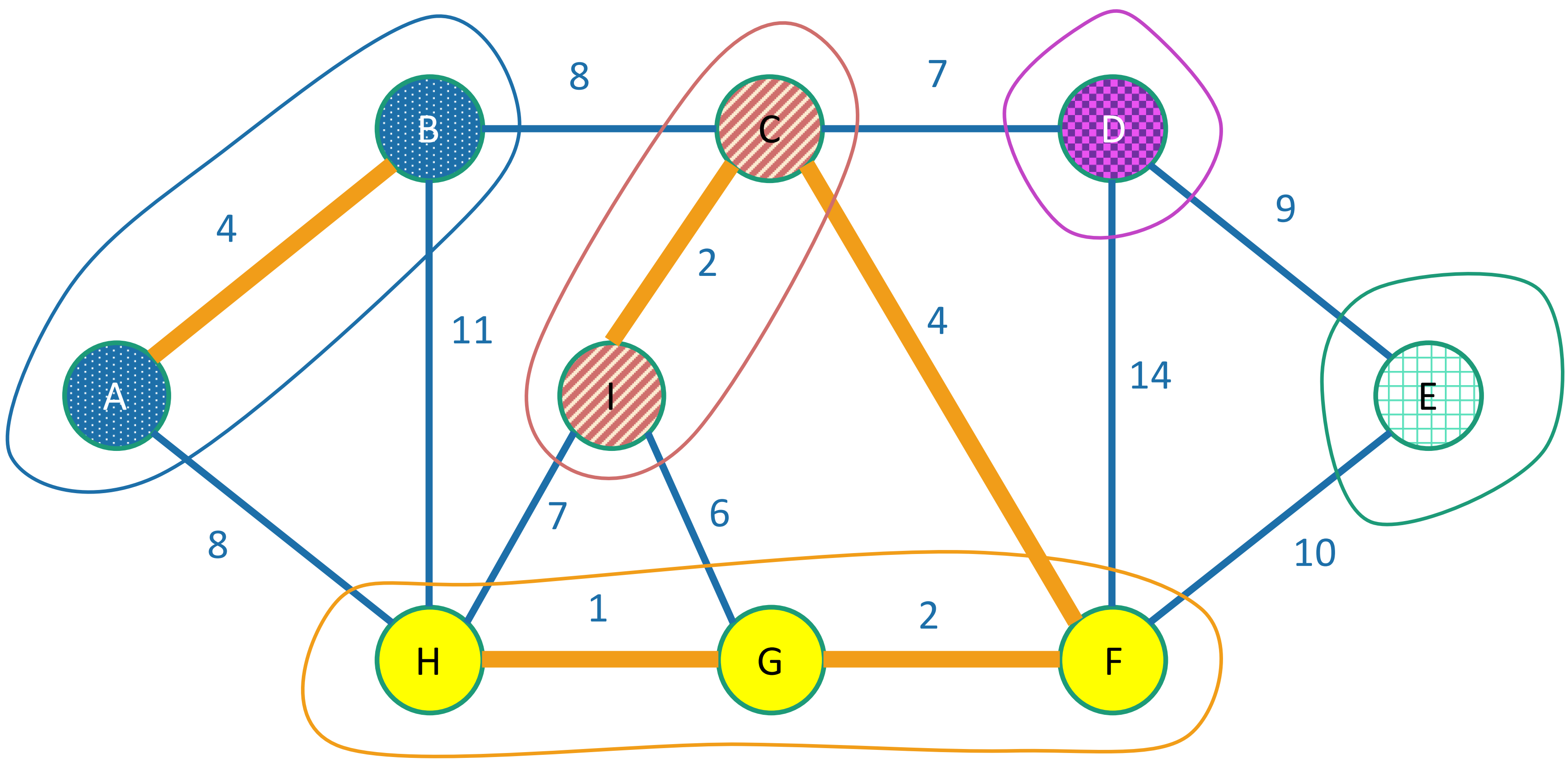



*Source: Stanford, CS 161 course, Winter 2022*

# At each step of Kruskal's

- We are maintaining a forest
- When we add an edge, we merge two trees

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

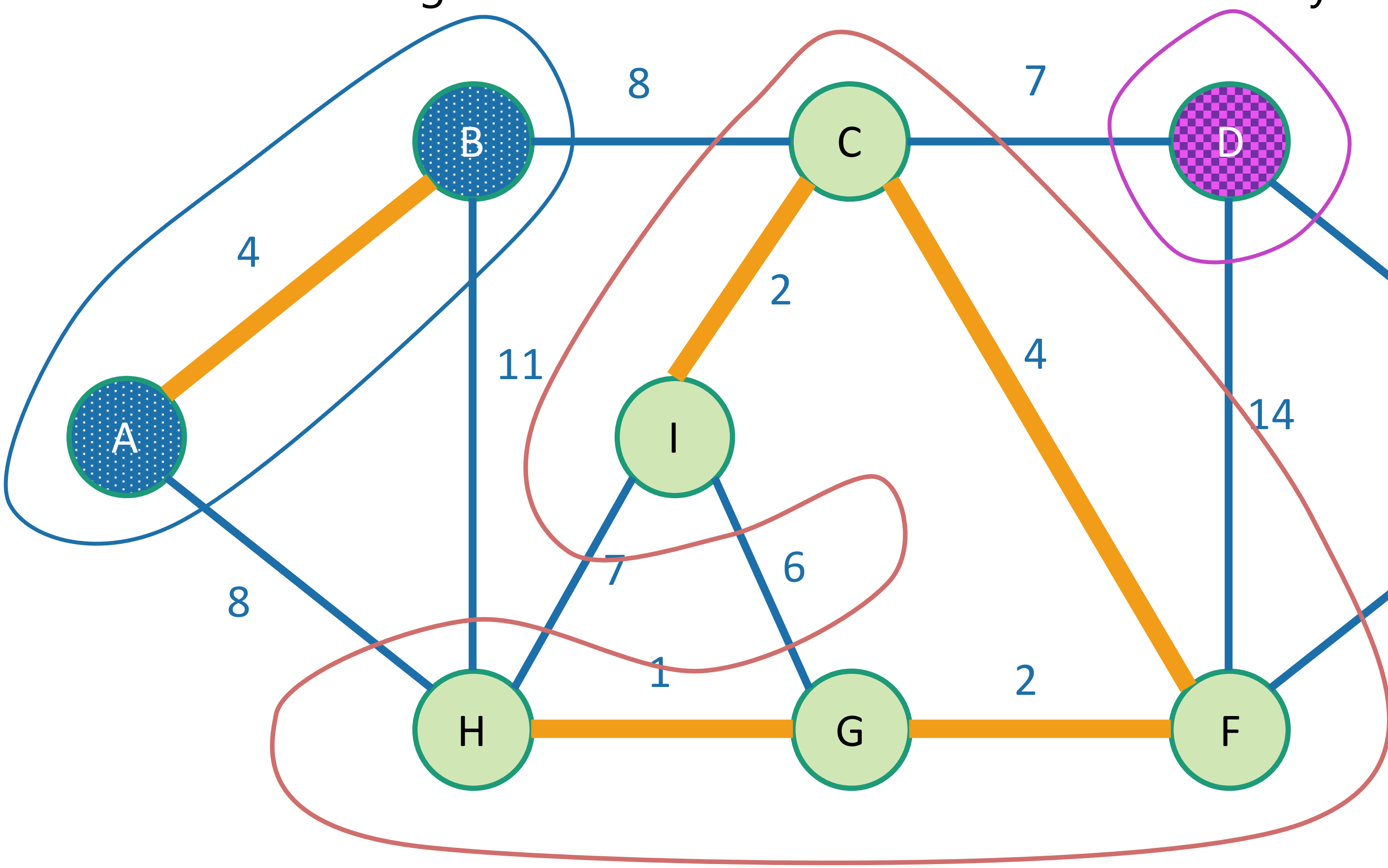



*Source: Stanford, CS 161 course, Winter 2022*

# 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

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




Source: Stanford, CS 161 course, Winter 2022

# Keep the Trees in a Special Data Structure

"treehouse"?

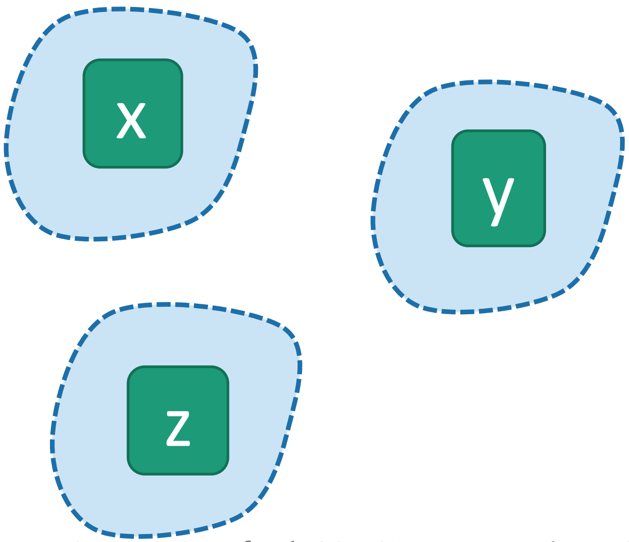
*Source: Stanford, CS 161 course, Winter 2022*

# 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)
```



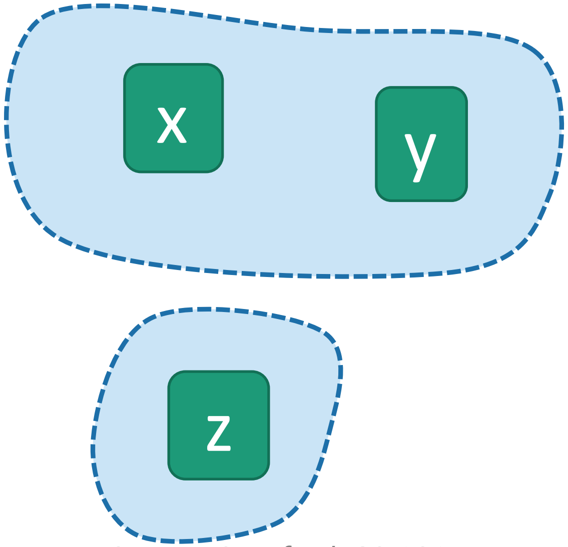

*Source: Stanford, CS 161 course, Winter 2022*

# 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)
```




*Source: Stanford, CS 161 course, Winter 2022*

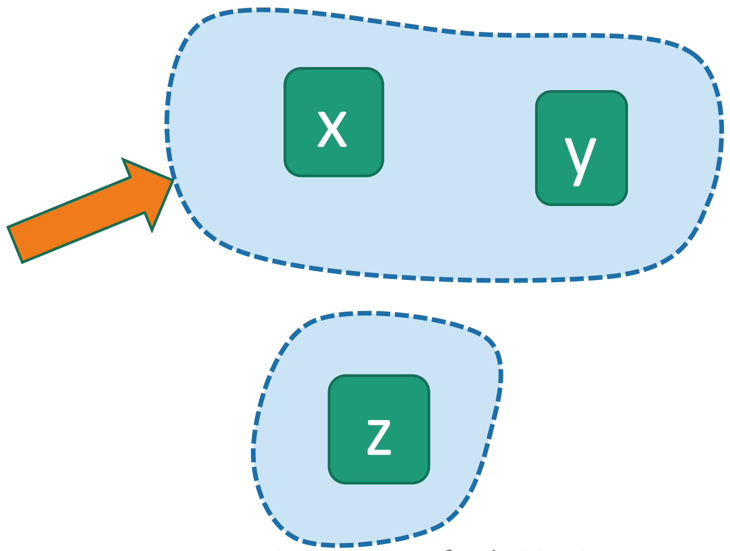
# Union-find Data Structure

- 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)
```




*Source: Stanford, CS 161 course, Winter 2022*

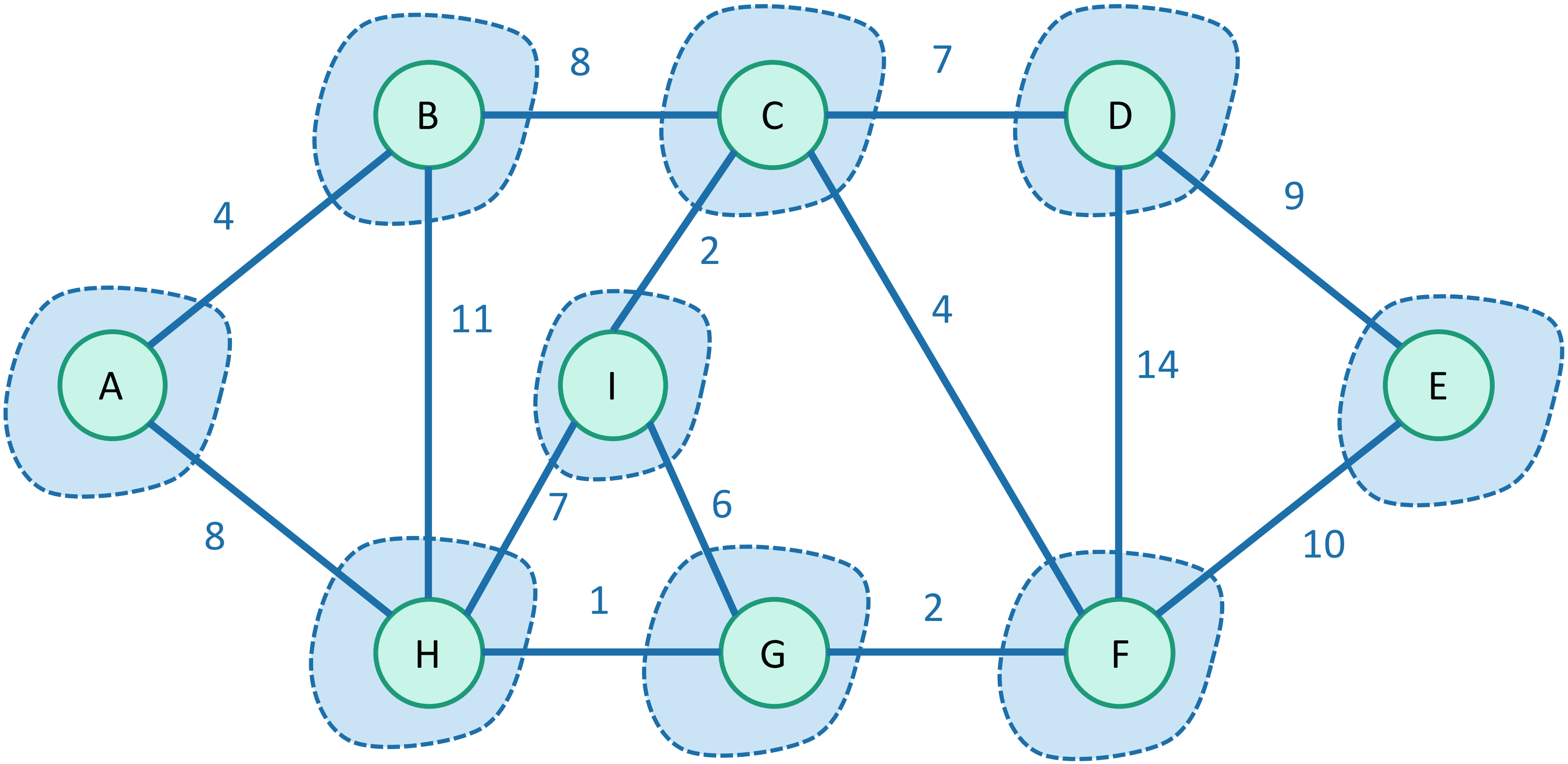
# Kruskal Pseudocode

- **Kruskal**(G = (V,E)):
  - Sort the edges in E by non-decreasing weight.
  - 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

*Source: Stanford, CS 161 course, Winter 2022*

# Once More …

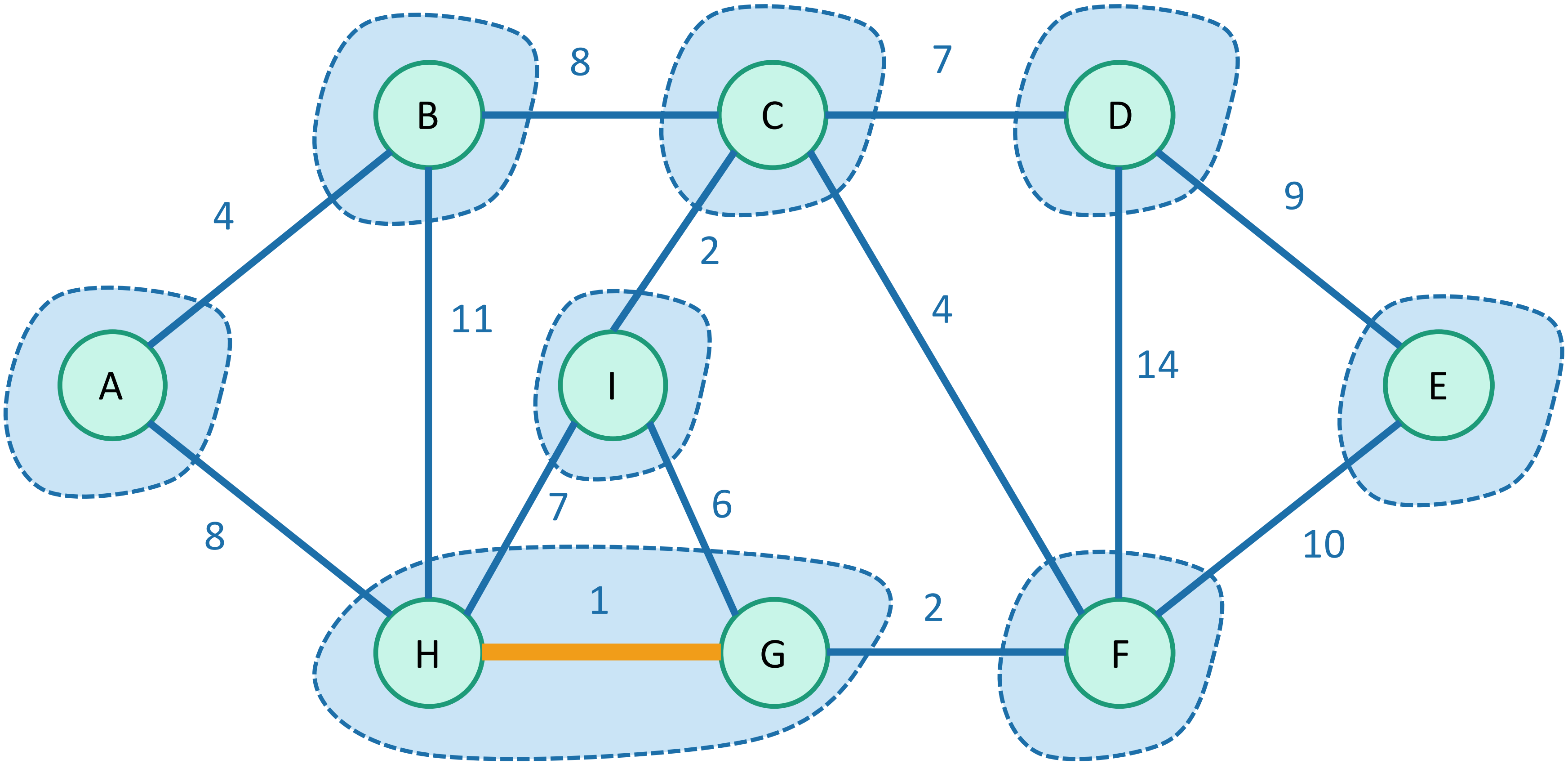
- To start, every vertex is in its own tree




*Source: Stanford, CS 161 course, Winter 2022*

# Once More …

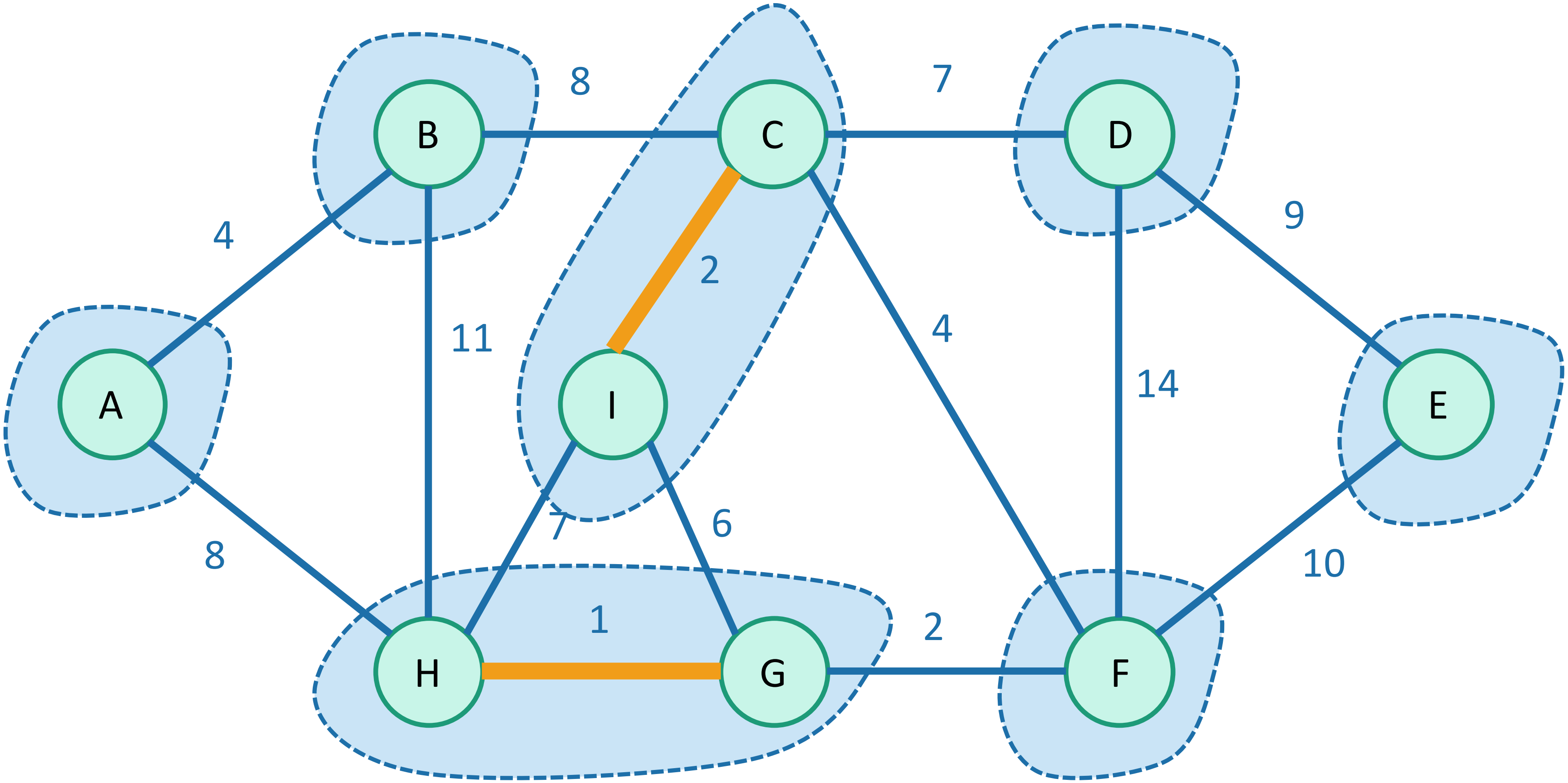
- Then start merging




Source: Stanford, CS 161 course, Winter 2022

# Once More …

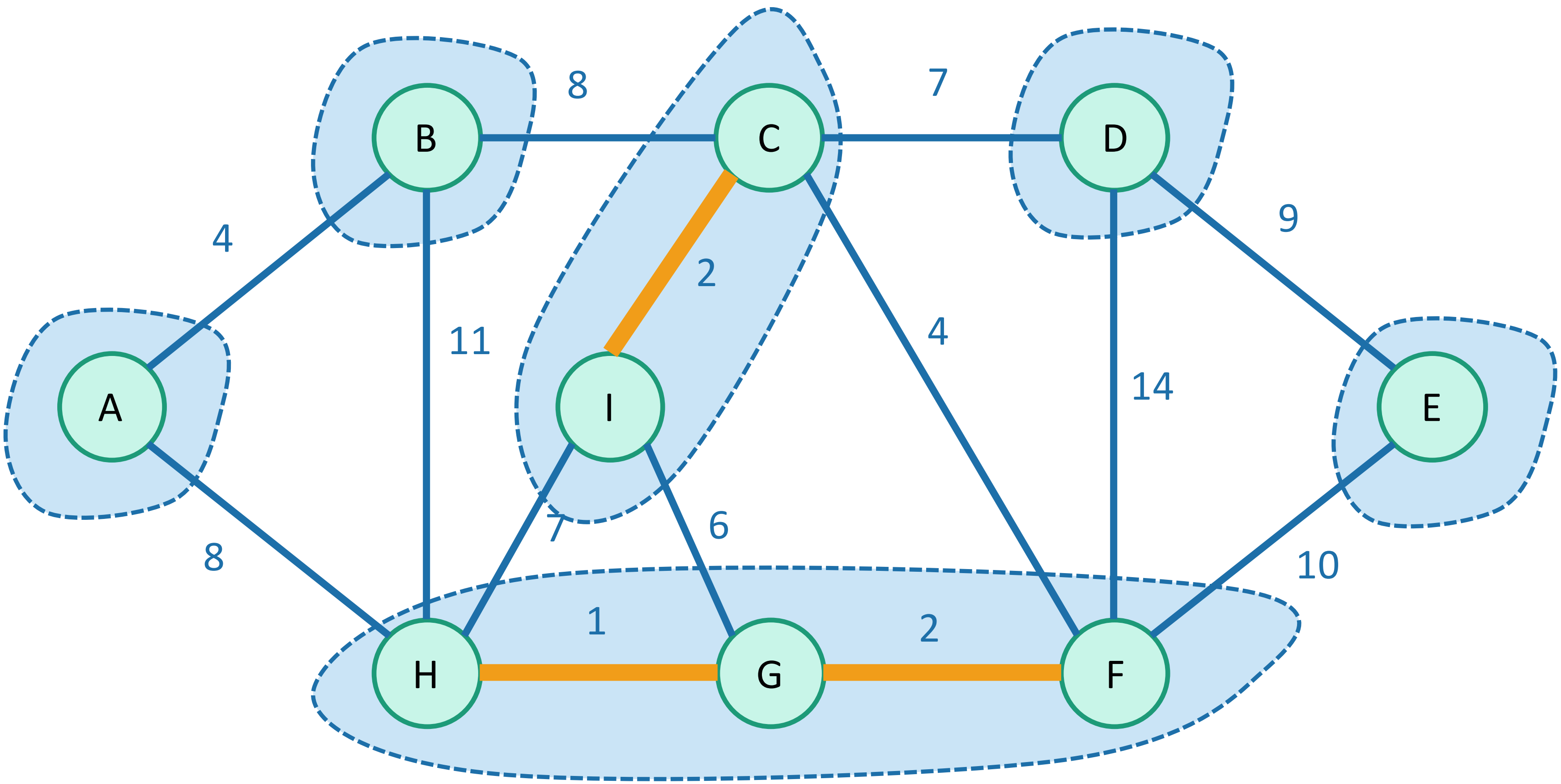
- Then start merging




Source: Stanford, CS 161 course, Winter 2022

# Once More …

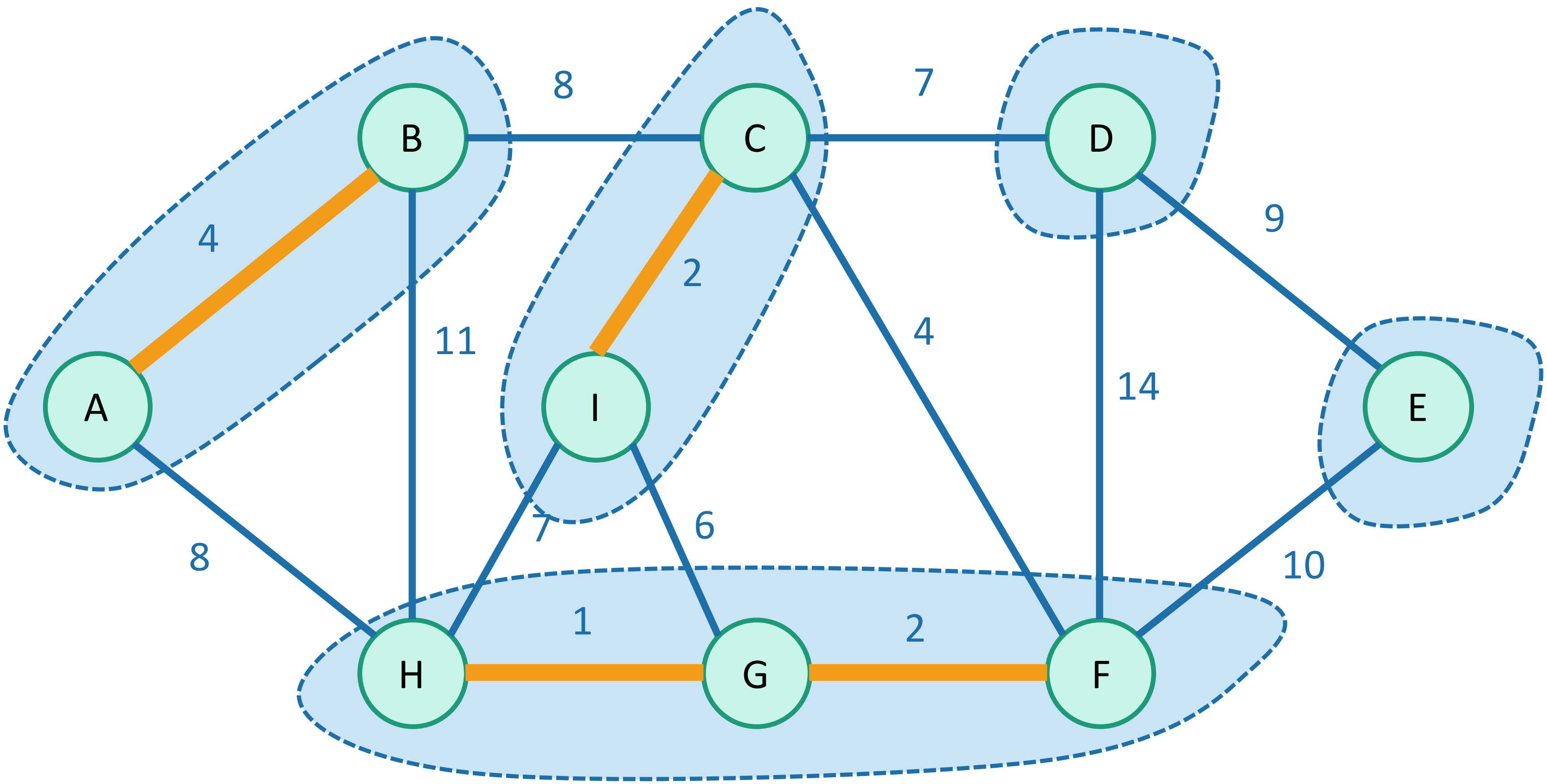
- Then start merging




*Source: Stanford, CS 161 course, Winter 2022*

# Once More …

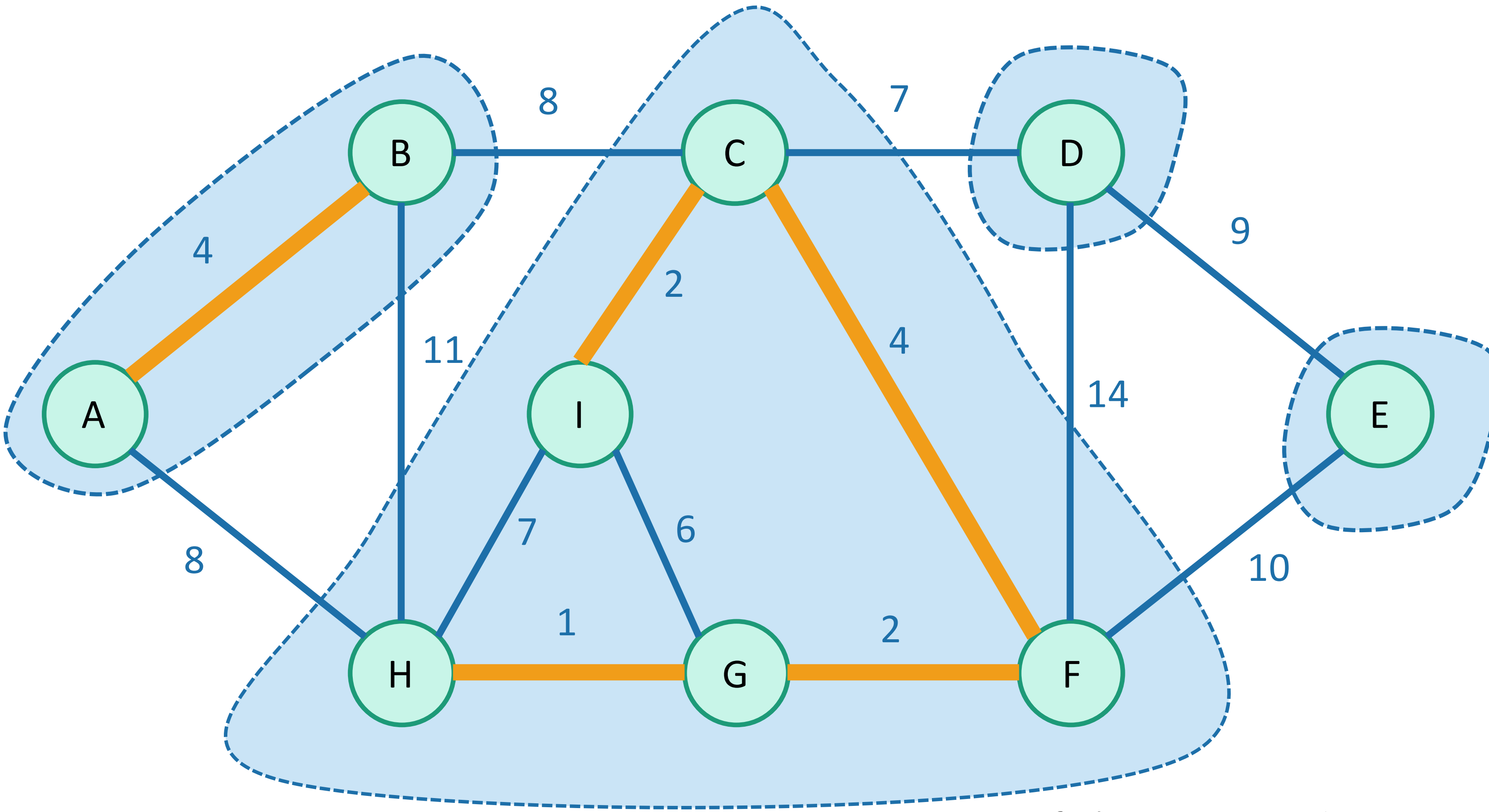
- Then start merging




Source: Stanford, CS 161 course, Winter 2022

# Once More …

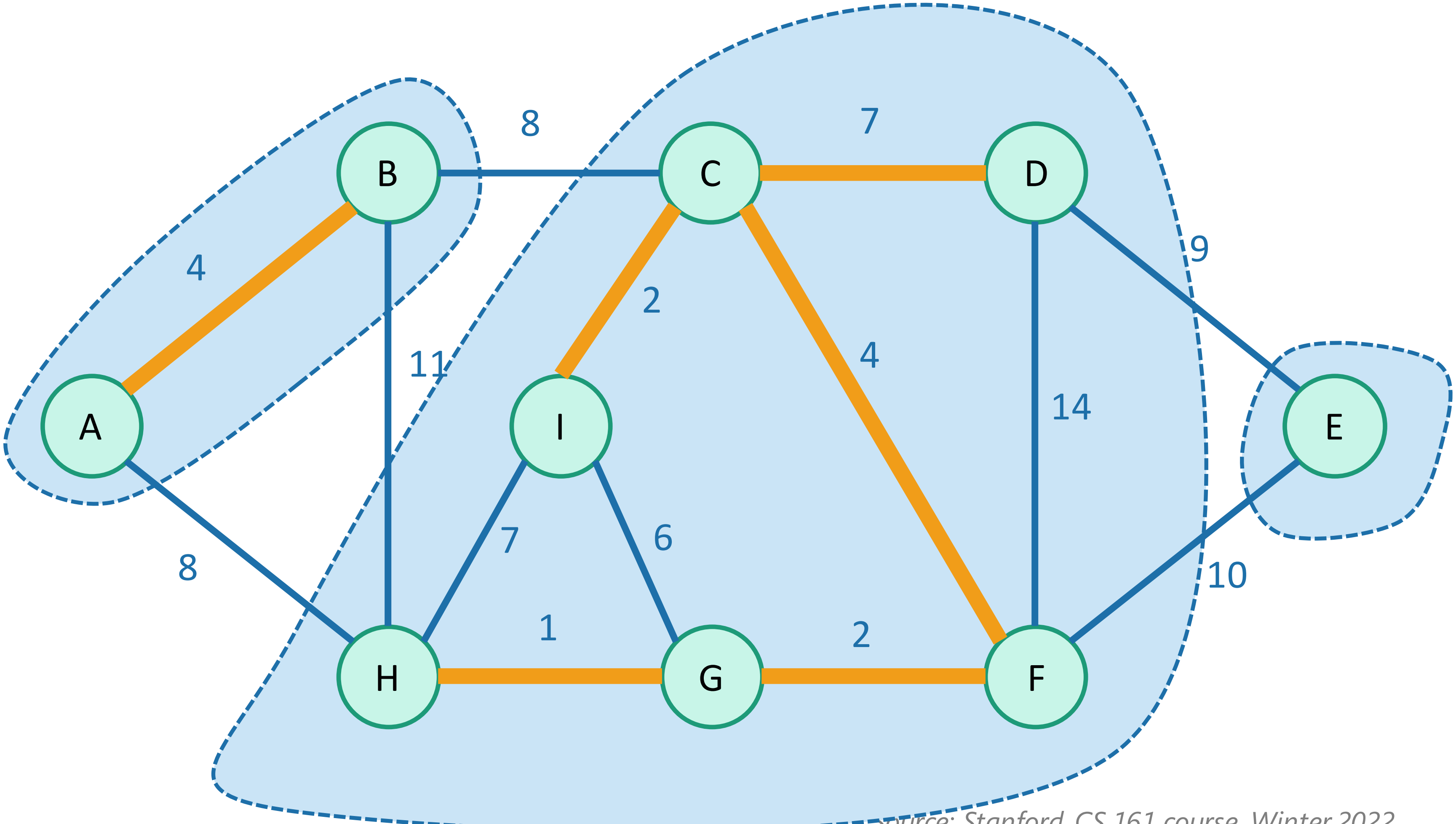
- Then start merging




Source: Stanford, CS 161 course, Winter 2022

# Once More ...

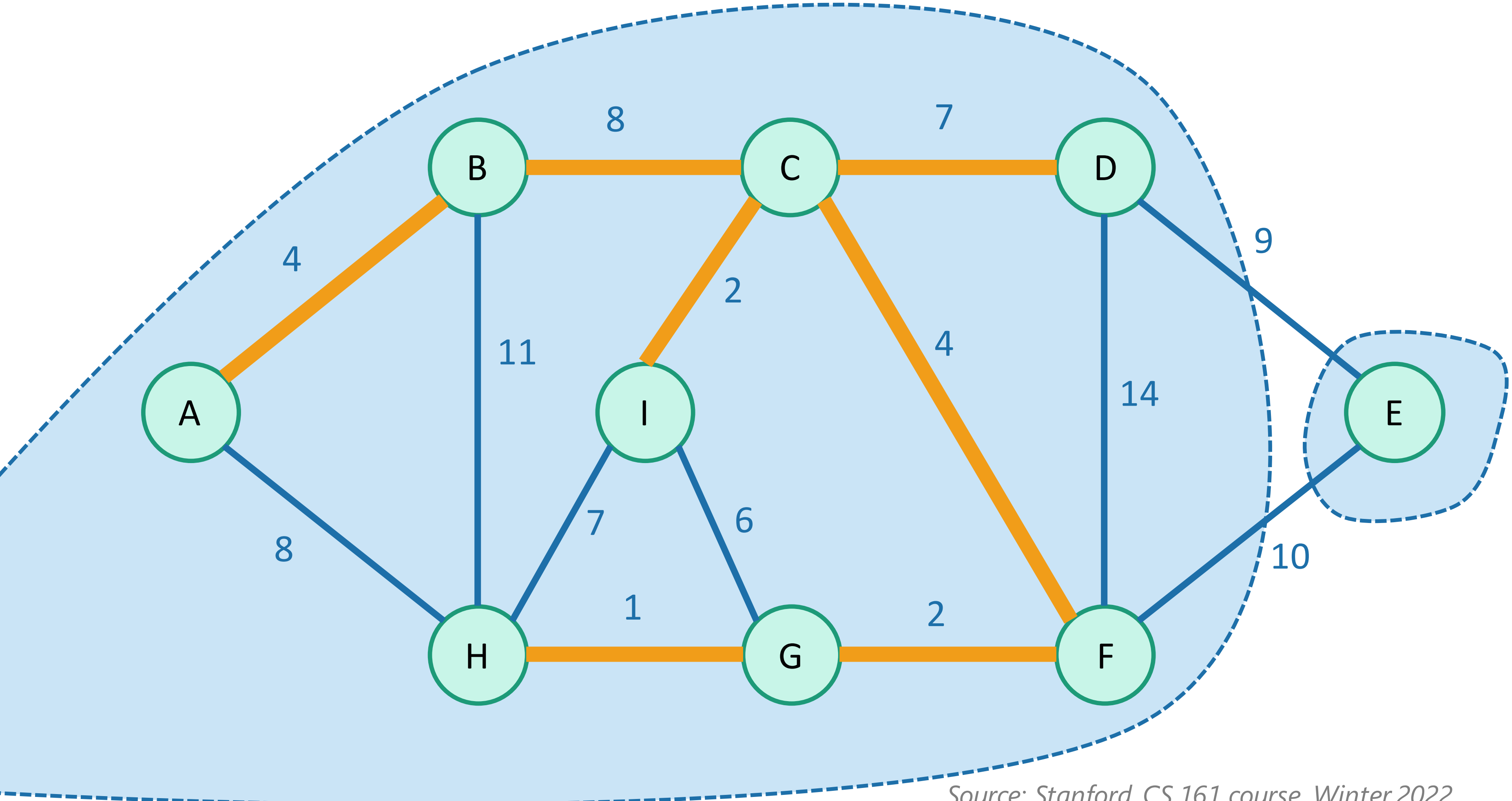
- Then start merging




Source: Stanford, CS 161 course, Winter 2022

# Once More …

- Then start merging

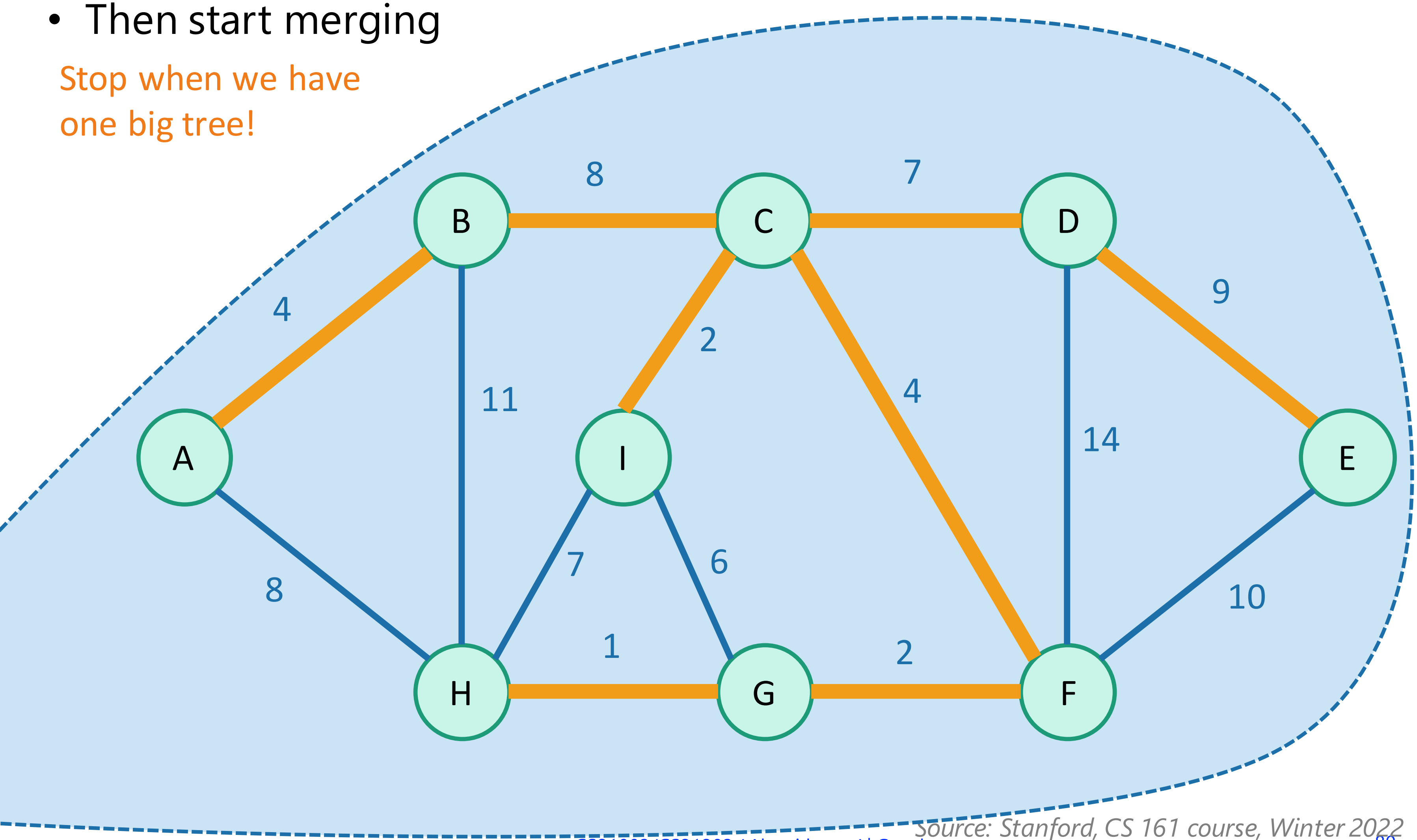



Source: Stanford, CS 161 course, Winter 2022

# Once More ...

- Then start merging

Stop when we have one big tree!




Source: Stanford, CS 161 course, Winter 2022

# Running Time

- Sorting the edges takes $O(|E| \log |V|)$
- For the rest
  - $|V|$ calls to **makeSet**
    - Put each vertex in its own set
  - $|2E|$ calls to **find**
    - For each edge, find its end points
  - $|V - 1|$ 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
- Total running time:
  - Worst-case $O(|E| \log |V|)$

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

# Two questions

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

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

- How do we actually implement this?
  - The pseudocode above says "slowKruskal" …
  - Worst-case running time $O(|E| \log |V|)$ using a union-find data structure

*Source: Stanford, CS 161 course, Winter 2022*

# Does it Work?

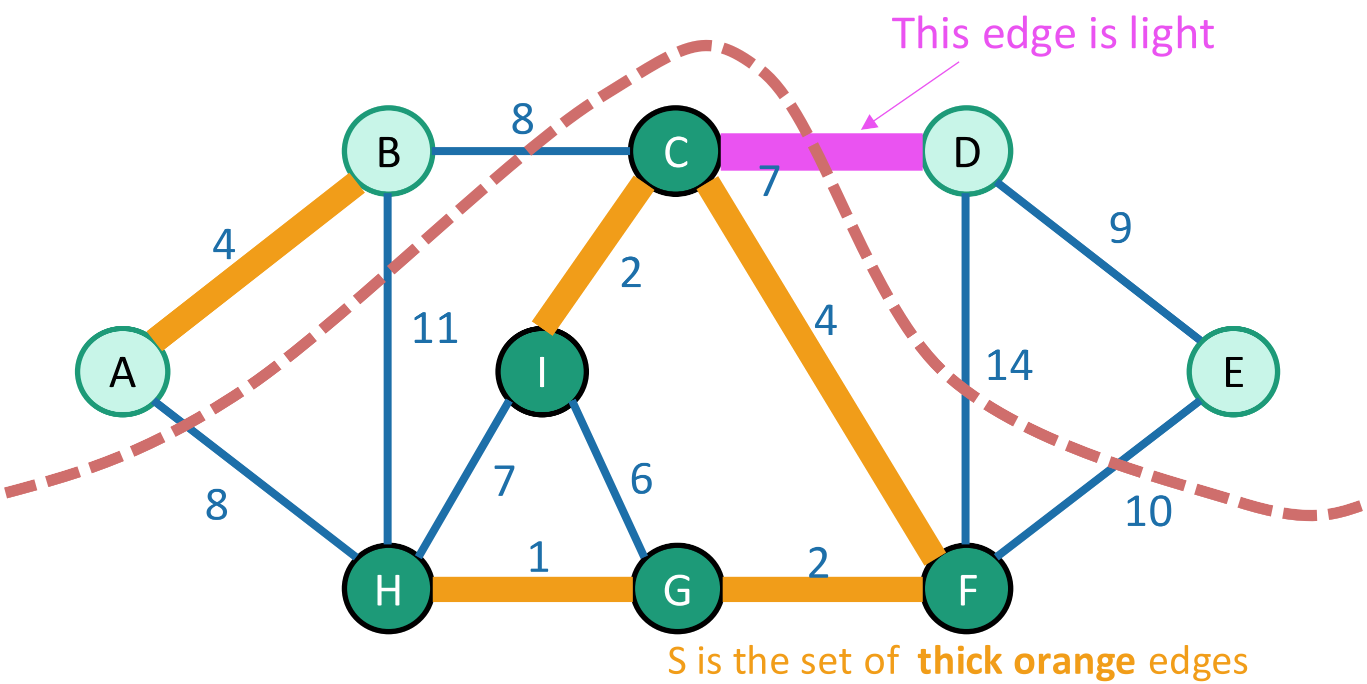
- We need to show that our greedy choices don't rule out success
- That is, at every step:
  - There exists an MST that contains all of the edges we have added so far
- Now it is time to use our lemma!

**again!**

*Source: Stanford, CS 161 course, Winter 2022*

# 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}}

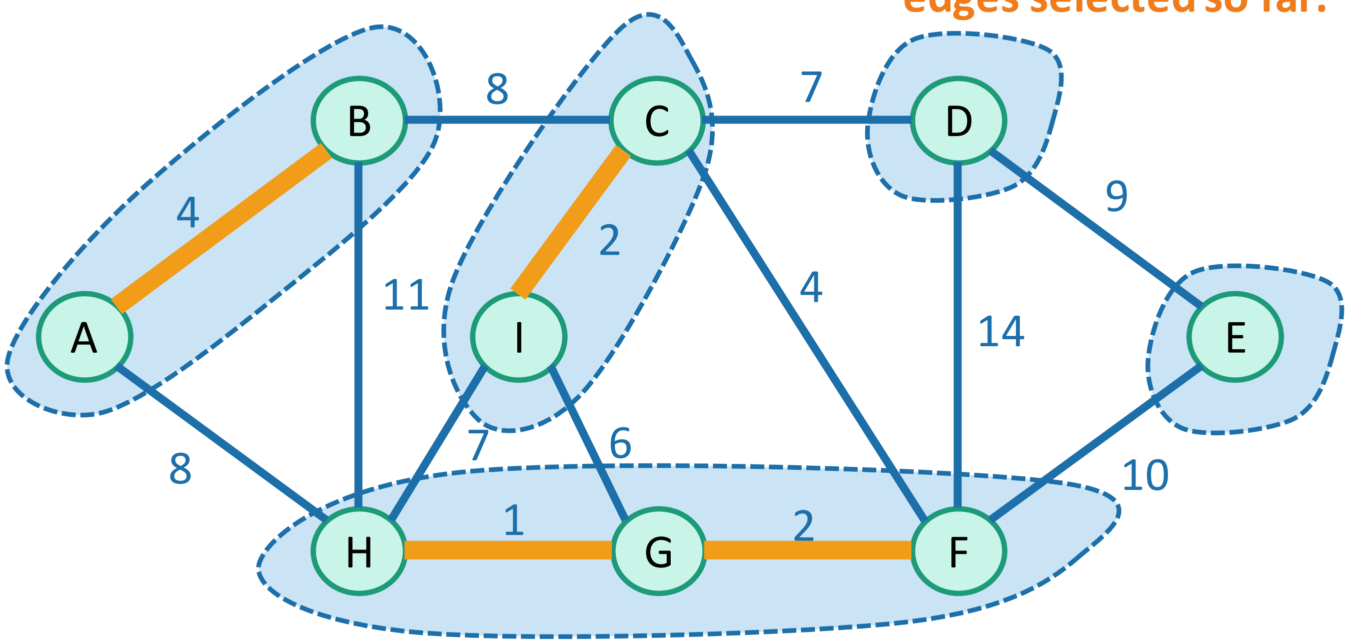



Source: Stanford, CS 161 course, Winter 2022

# Partway through Kruskal

- Assume that our choices S so far don’t rule out success
  - There is an MST extending them

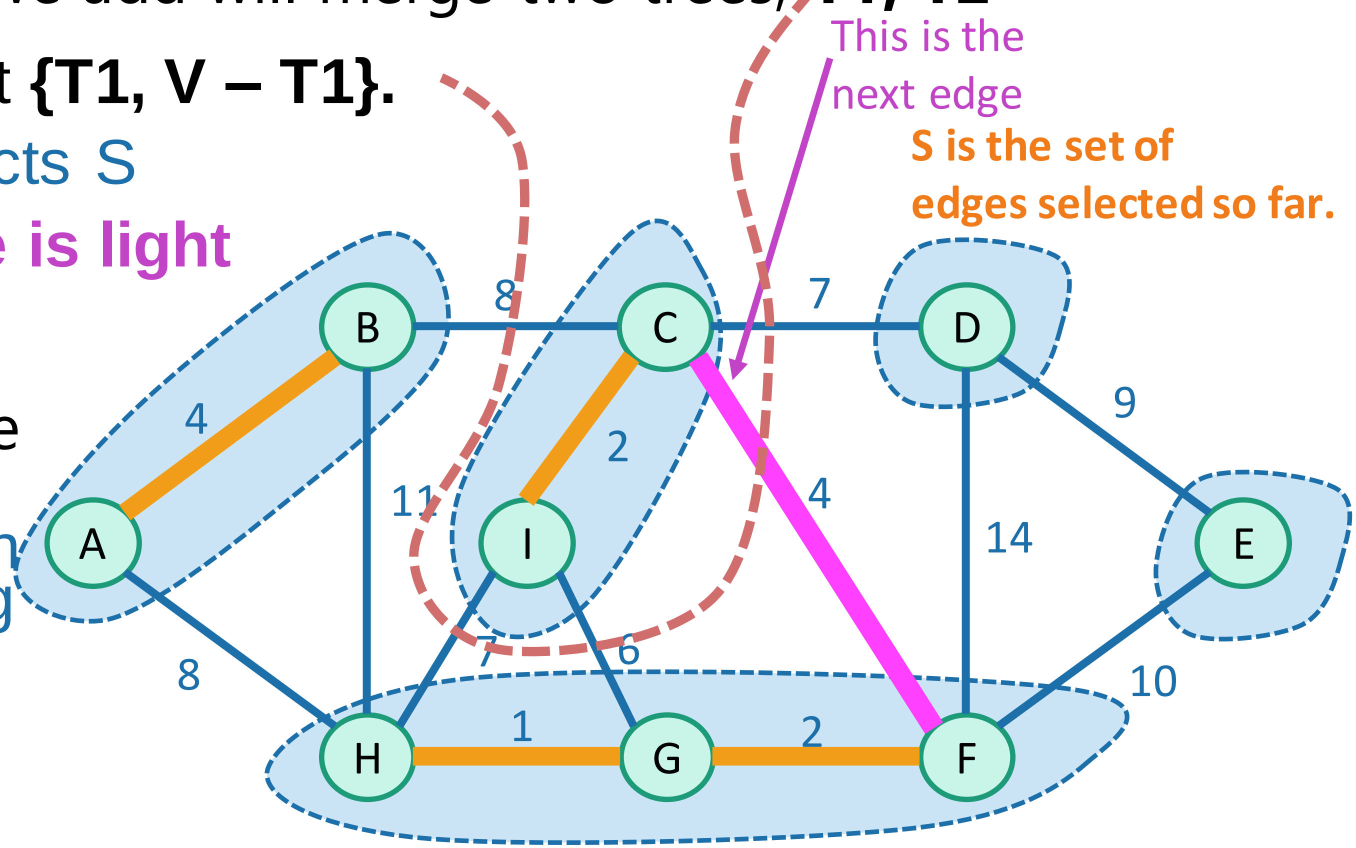
**S is the set of edges selected so far.**




Source: Stanford, CS 161 course, Winter 2022

# 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




Source: Stanford, CS 161 course, Winter 2022

# Partway through Kruskal

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

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

Source: Stanford, CS 161 course, Winter 2022

# Two questions

- Does it work?
  - That is, does it actually return a MST?
  - Yes
- How do we actually implement this?
  - The pseudocode above says "slowKruskal" …
  - Using a union-find data structure!

*Source: Stanford, CS 161 course, Winter 2022*

# Recap

- Two algorithms for Minimum Spanning Tree
  - Prim's algorithm
  - Kruskal's algorithm

- Both are 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

*Source: Stanford, CS 161 course, Winter 2022*

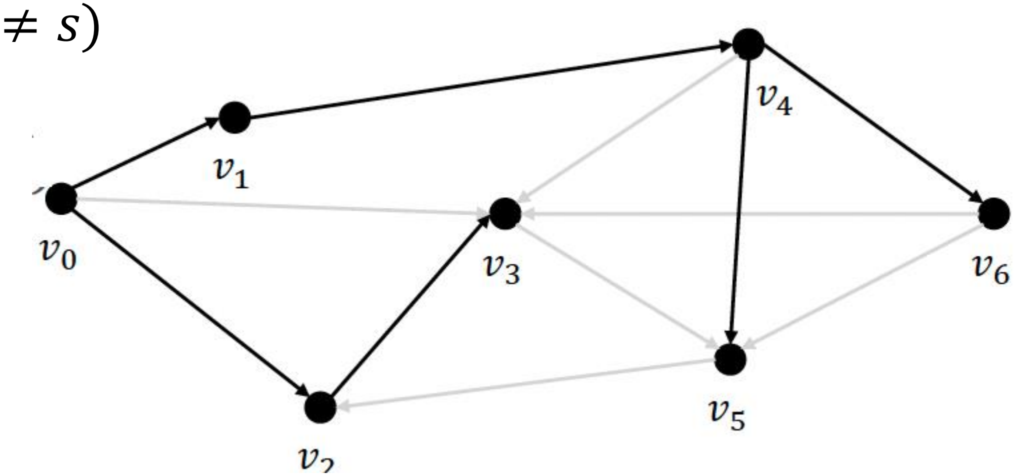
# SSSP Again

- We have seen Dijkstra’s method
  - One drawback is that it needs non-negative edge weights

- Bellman-Ford algorithm
  - It is a dynamic programming algorithm
  - It has a higher cost than Dijkstra, but can handle graphs with negative edge weights

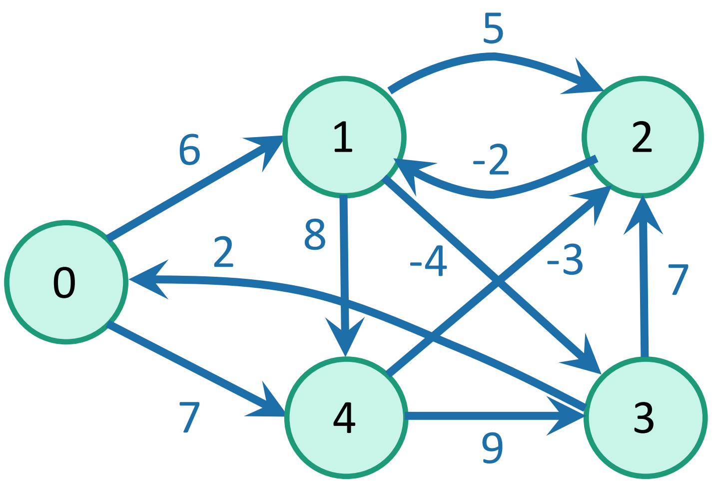
# Bellman-Ford as DP

- Let $D_{i,k}$ indicate the shortest distance from source $s$ to vertex $i$ using no more than $k$ hops (number of edges)
- Consider the last edge:
- $D_{i,k} = \min \begin{cases} D_{i,k-1} \\ \min\limits_{(j,i)\in E} \{D_{j,k-1} + w(j,i)\} \end{cases}$
- Boundaries: $D_{s,0} = 0, D_{i,0} = \infty \ (i \neq s)$
- Final answer to vertex $i$ is $D_{i,n-1}$




Source: UCR, CS 141 course, Fall 2021

# SSSP Again



# Thank You