香港科技大学(广州)
THE HONG KONG
UNIVERSITY OF SCIENCE AND
TECHNOLOGY (GUANGZHOU)

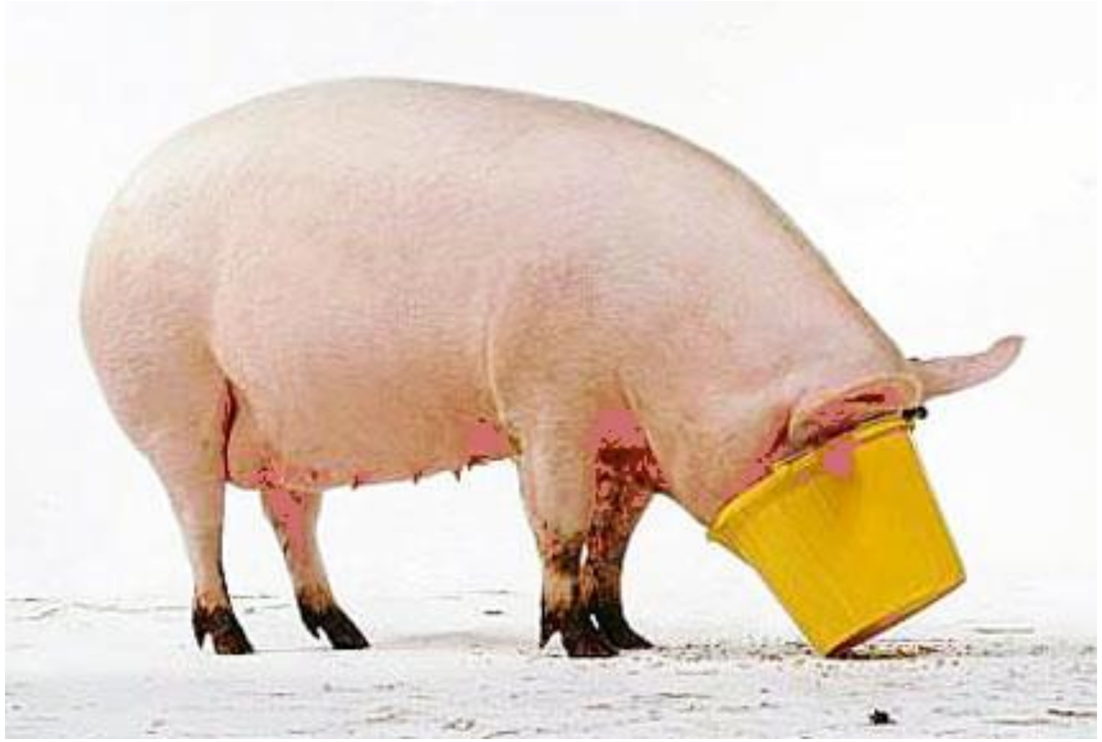
Design and Analysis of Algorithms

Jing Tang | DSAA 2043 Fall 2024

Greedy Algorithms

Greedy Algorithms

- Make choices one-at-a-time.
- Never look back.
- Hope for the best.



Today

One example of a greedy algorithm that **does not work**:

- Knapsack again

Three examples of greedy algorithms that **do work**:

- Activity Selection
- Job Scheduling
- Minimum Spanning Tree

Non-example: Unbounded Knapsack

Capacity: 10

Item:					
Weight:	6	2	4	3	11
Value:	20	8	14	13	35

- Unbounded Knapsack:
  - Suppose I have infinite copies of all items.
  - What's the most valuable way to fill the knapsack?

Total weight: 10
Total value: 42

- "Greedy" algorithm for unbounded knapsack:
  - Tacos have the best Value/Weight ratio!
  - Keep grabbing tacos!

Total weight: 9
Total value: 39

Example where greedy works

DSAA1001 Class

Play Games

UFUG2103 Class

Sleep

DSAA2043

Frisbee Practice

inar

Progran
team m

43 Class

You can only do one activity at a time, and you want to maximize the number of activities that you do.

What to choose?

weaving class

DSAA2043
study group

Swimming
lessons

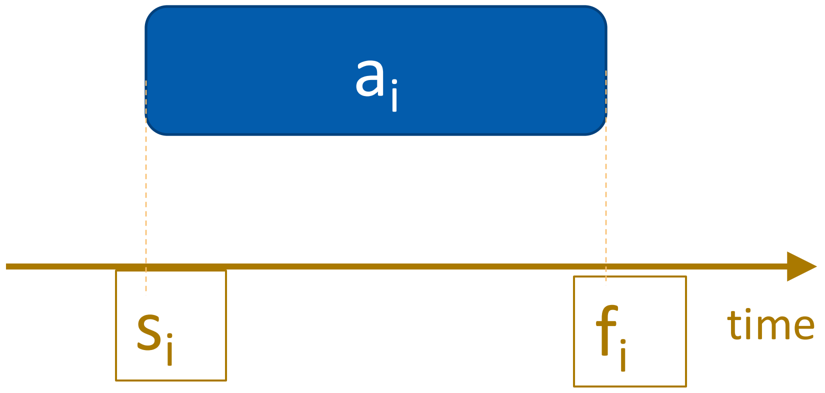
Research
Project

Lunch Salon

Social
activity

time

Activity selection

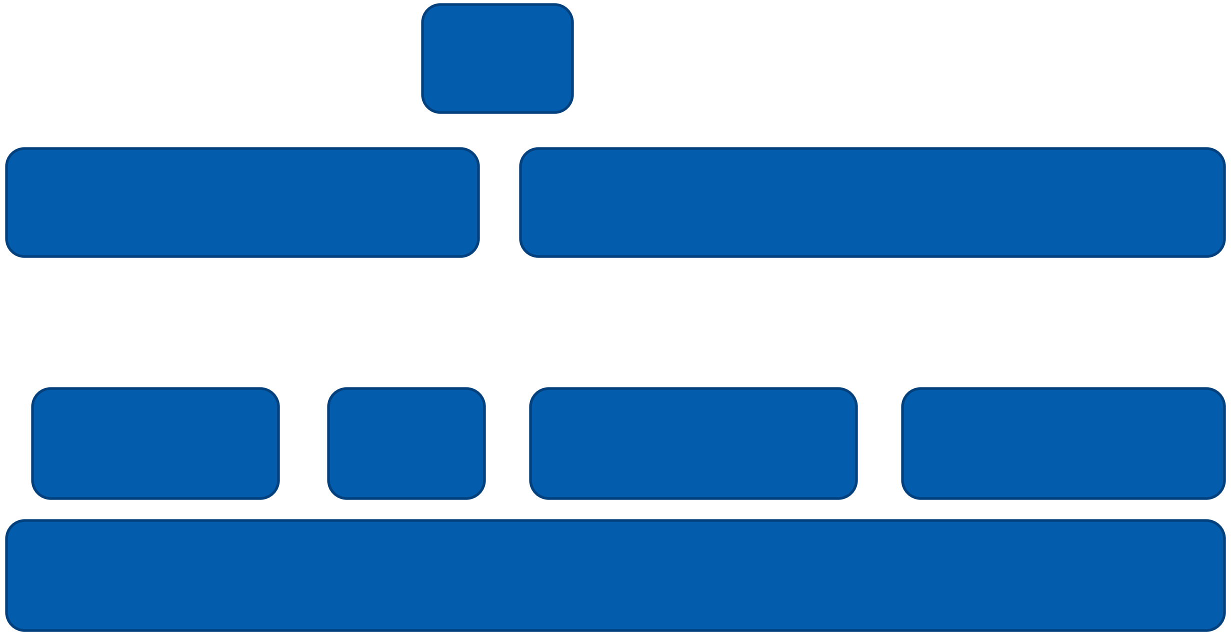


- Input:
  - Activities $a_1, a_2, \ldots, a_n$
  - Start times $s_1, s_2, \ldots, s_n$
  - Finish times $f_1, f_2, \ldots, f_n$
- Output:
  - A way to maximize the number of activities you can do today.

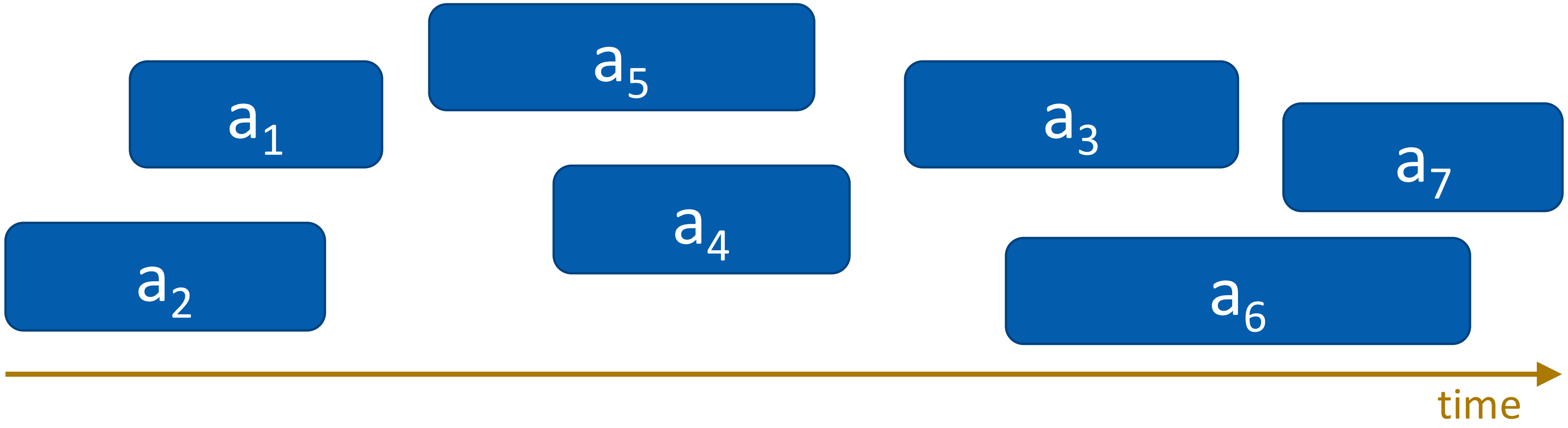
In what order should you greedily add activities?

In what order?

- Shortest job first?
- Earliest start time?
- Earliest finish time?

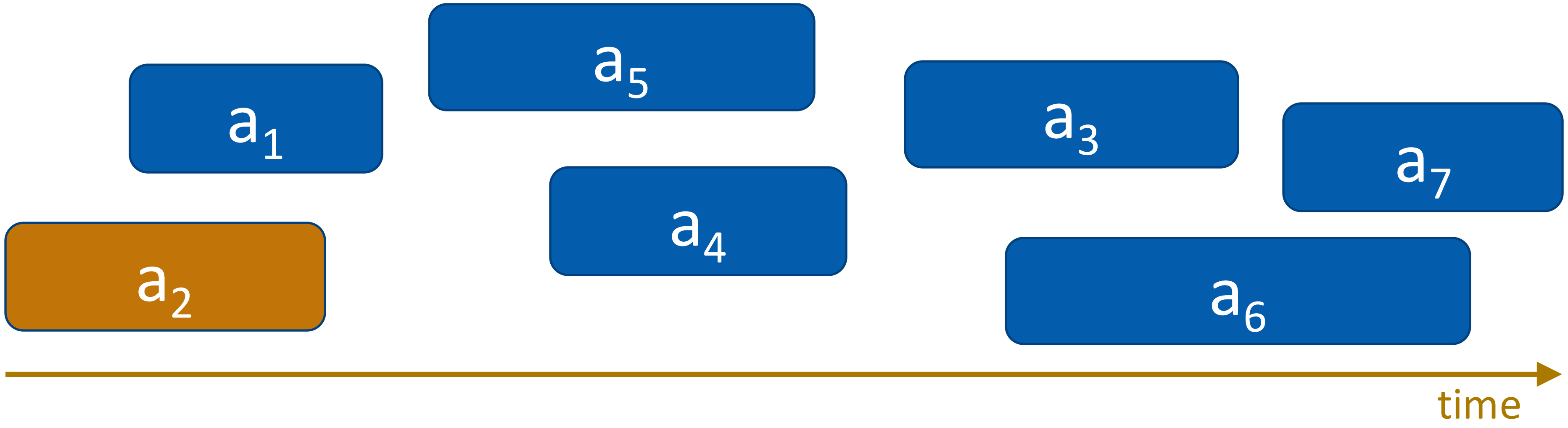


Greedy Algorithm



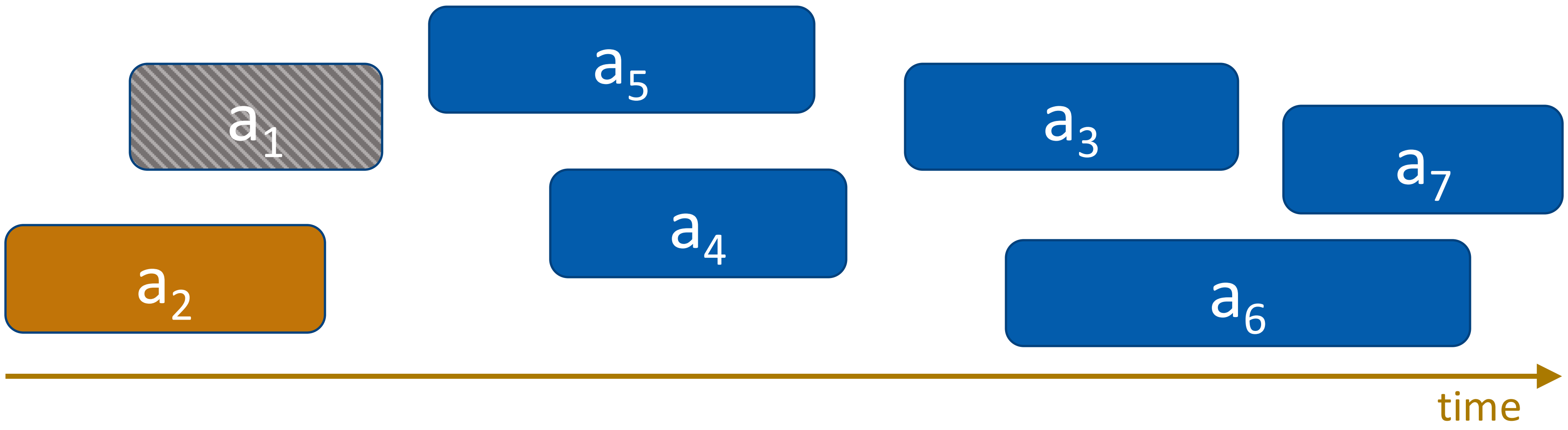
- Pick activity you can add with the smallest finish time.
- Repeat.

Greedy Algorithm



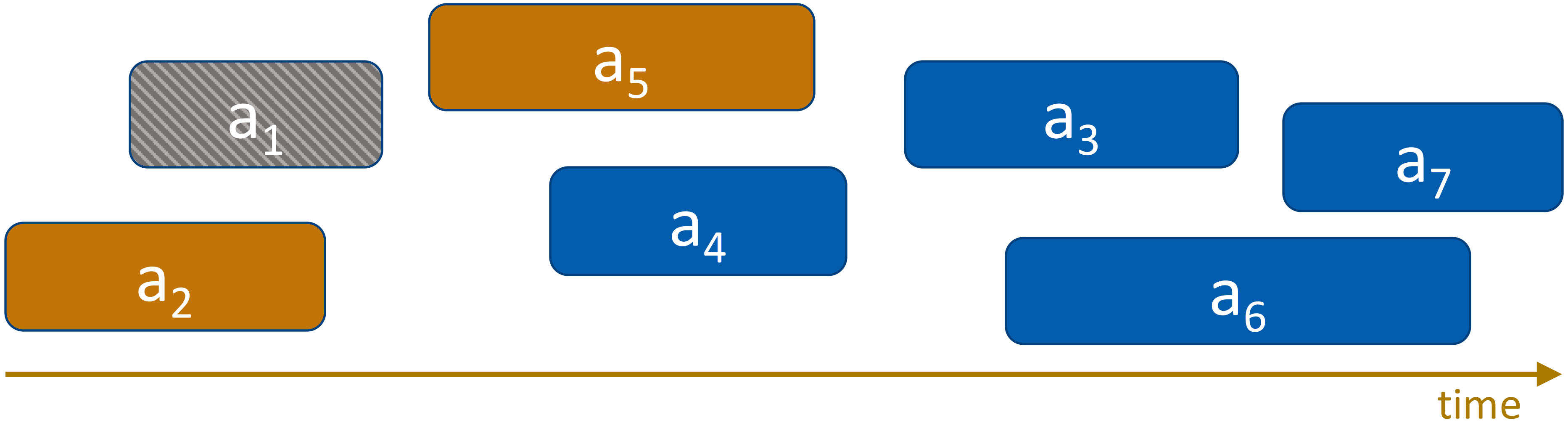
- Pick activity you can add with the smallest finish time.
- Repeat.

Greedy Algorithm



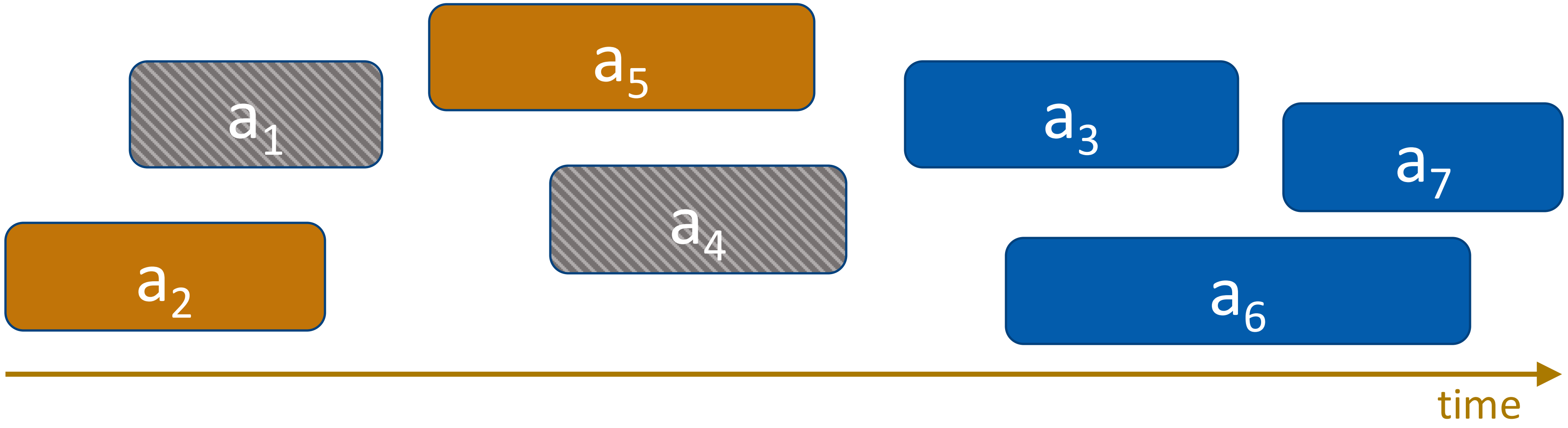
- Pick activity you can add with the smallest finish time.
- Repeat.

Greedy Algorithm



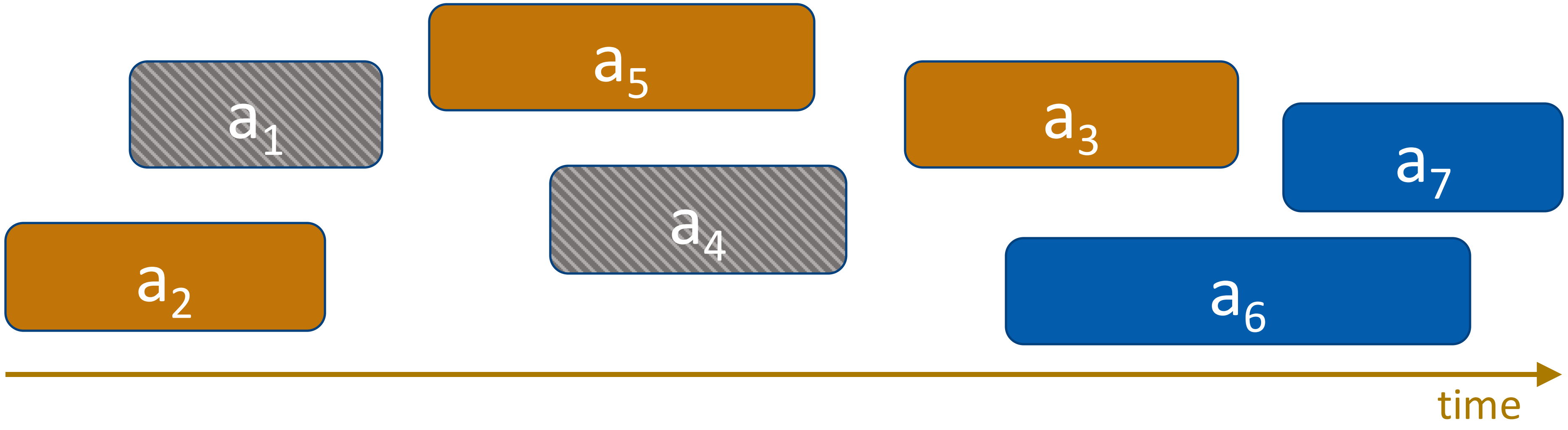
- Pick activity you can add with the smallest finish time.
- Repeat.

Greedy Algorithm



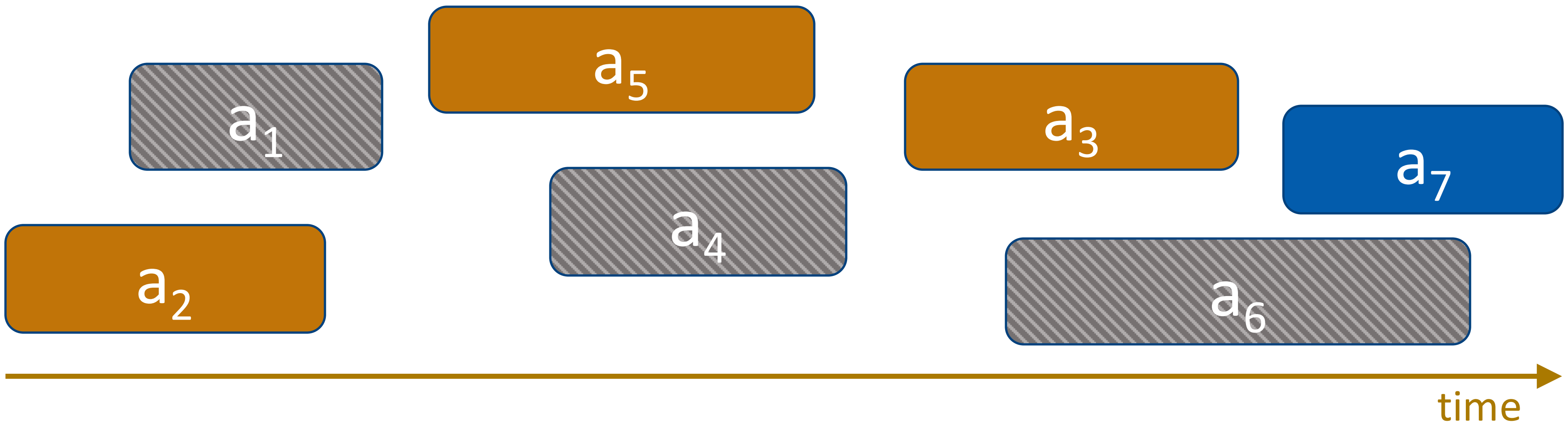
- Pick activity you can add with the smallest finish time.
- Repeat.

Greedy Algorithm



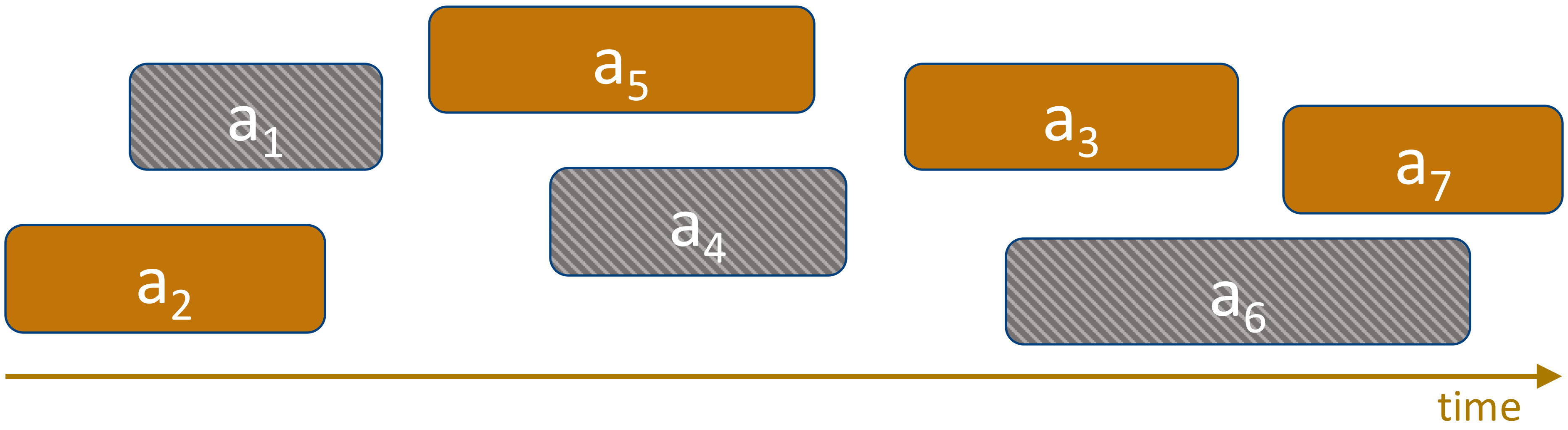
- Pick activity you can add with the smallest finish time.
- Repeat.

Greedy Algorithm



- Pick activity you can add with the smallest finish time.
- Repeat.

Greedy Algorithm



- Pick activity you can add with the smallest finish time.
- Repeat.

Efficiency

- Running time:
  - O(n) if the activities are already sorted by finish time.
  - Otherwise, O(n log(n)) if you have to sort them first.

Three Questions

1. Does this greedy algorithm for activity selection work?
   – Yes

2. Greedy is simple. But why are we getting to it in week 11 (not earlier)?
   – Proving that greedy algorithms work is often not so easy…

3. In general, when are greedy algorithms a good idea?
   – When the problem exhibits especially nice optimal substructure.

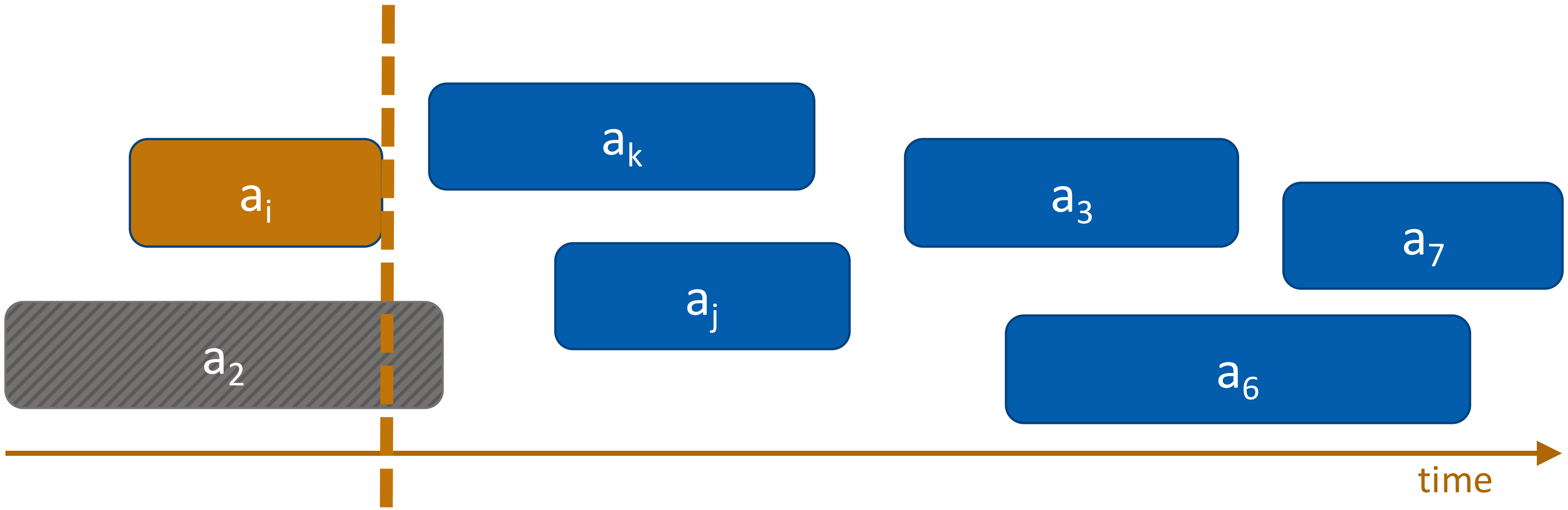
Back to Activity Selection

Why does it work?

- **We never rule out an optimal solution**
- At the end of the algorithm, we've got some solution.
- So it must be optimal.

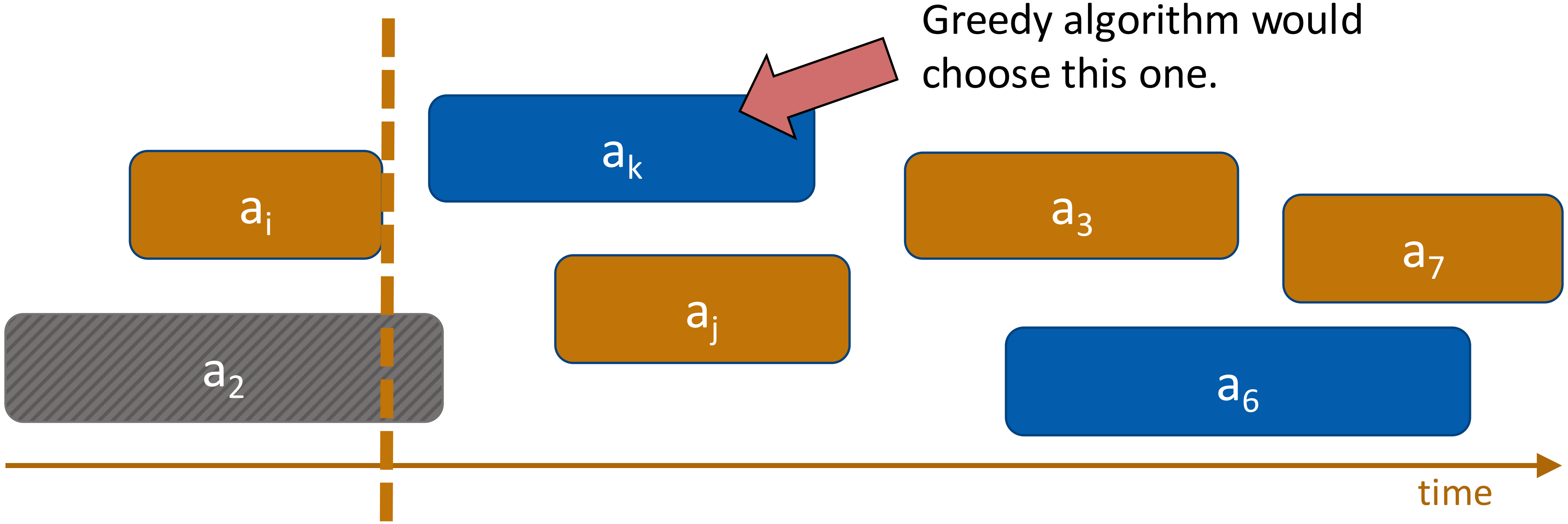
The Correctness of Activity Selection

- Suppose we've already chosen a_i, and there is still an optimal solution T* that extends our choices.



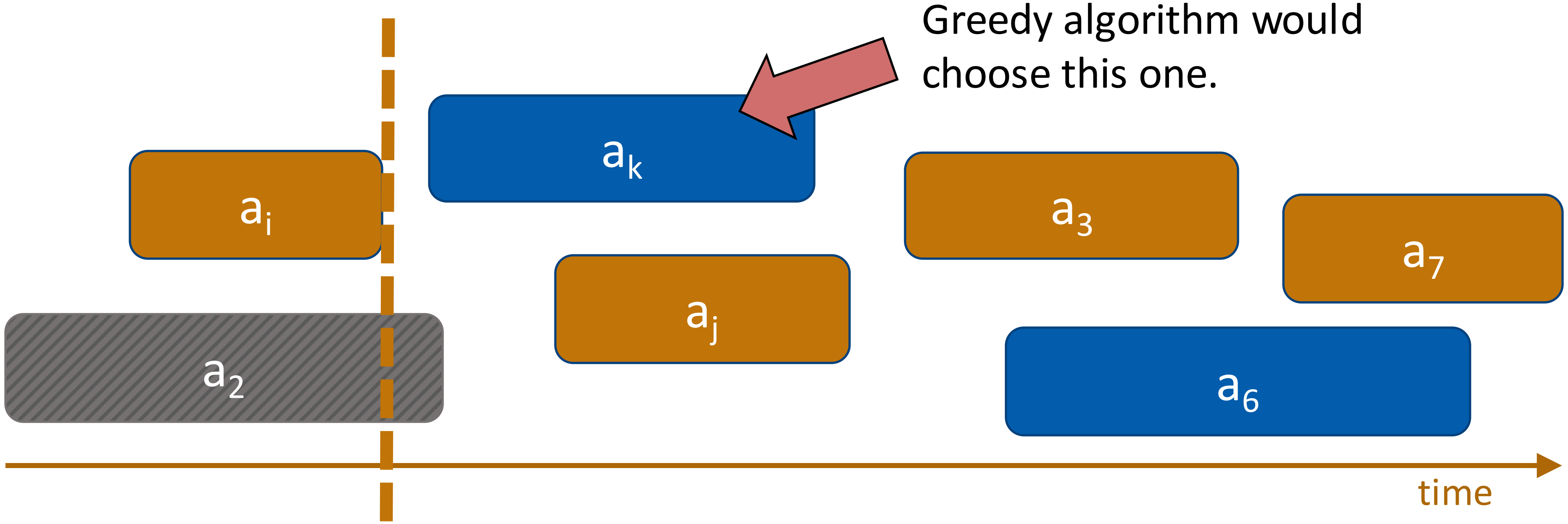
The Correctness of Activity Selection

- Suppose we've already chosen a_i, and there is still an optimal solution T* that extends our choices.
- Now consider the next choice we make, say it's a_k.
- If a_k is in T*, we're still on track.



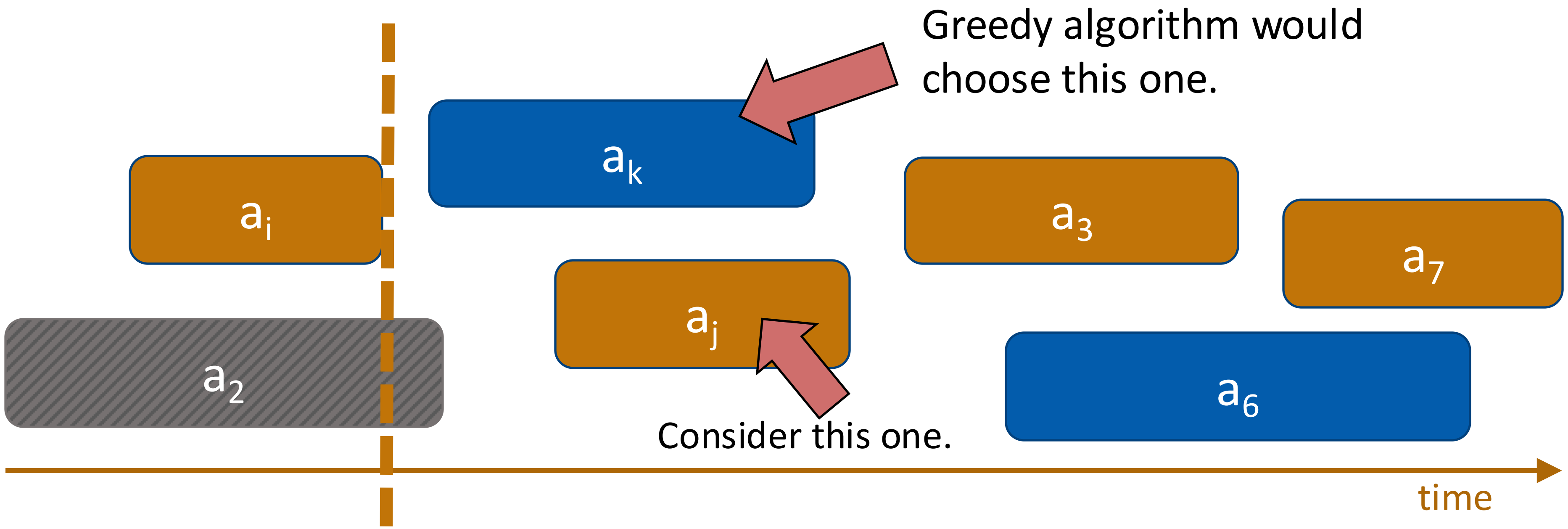
The Correctness of Activity Selection

- Suppose we've already chosen a_i, and there is still an optimal solution T* that extends our choices.
- Now consider the next choice we make, say it's a_k.
- If a_k is not in T*…



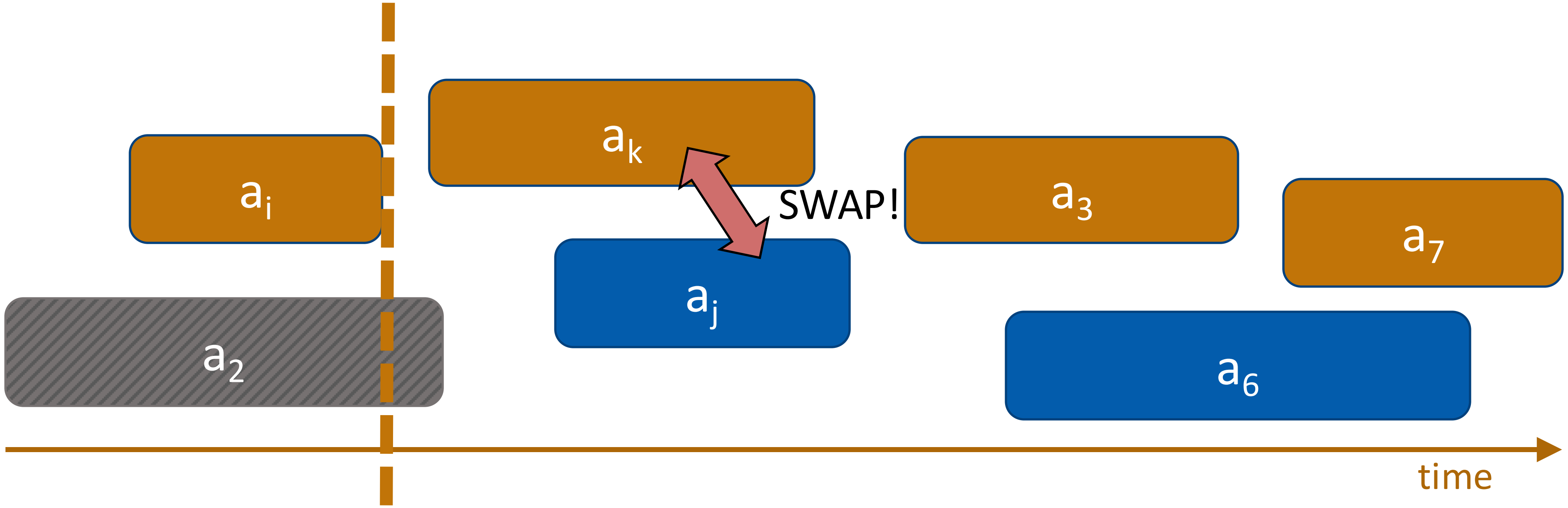
The Correctness of Activity Selection

- If a_k is **not** in T*…
- Let a_j be the activity in T* with the smallest end time.
- Now consider schedule T you get by swapping a_j for a_k



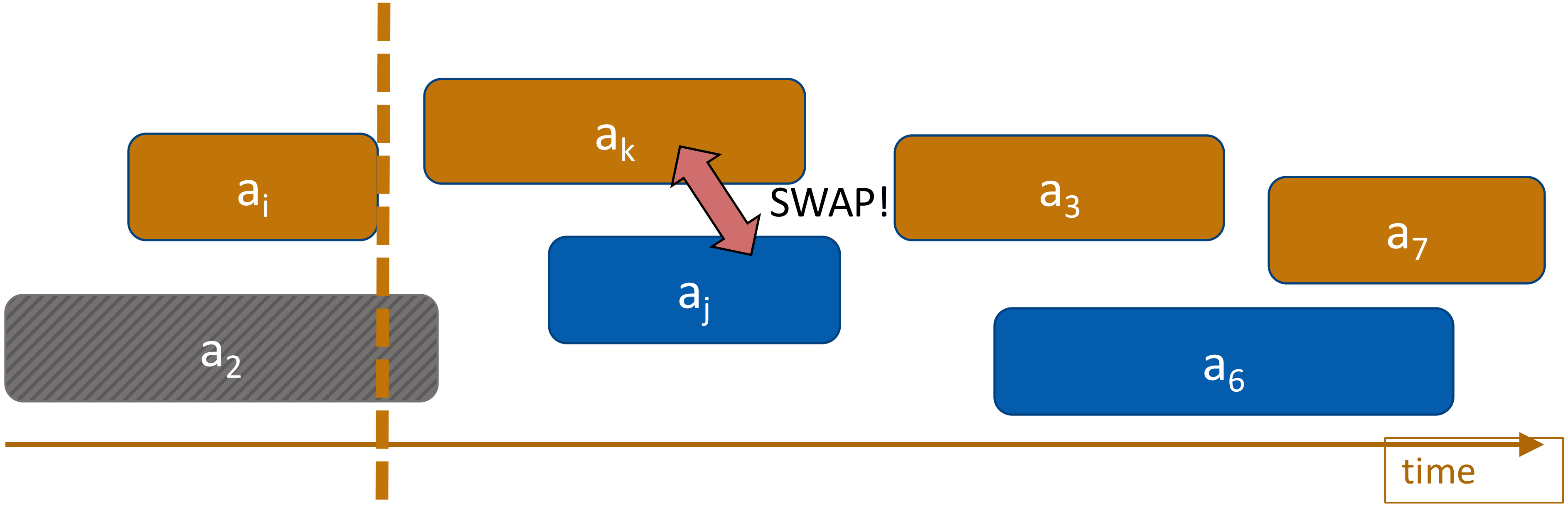
The Correctness of Activity Selection

- If a_k is **not** in T*…
- Let a_j be the activity in T* with the smallest end time.
- Now consider schedule T you get by swapping a_j for a_k



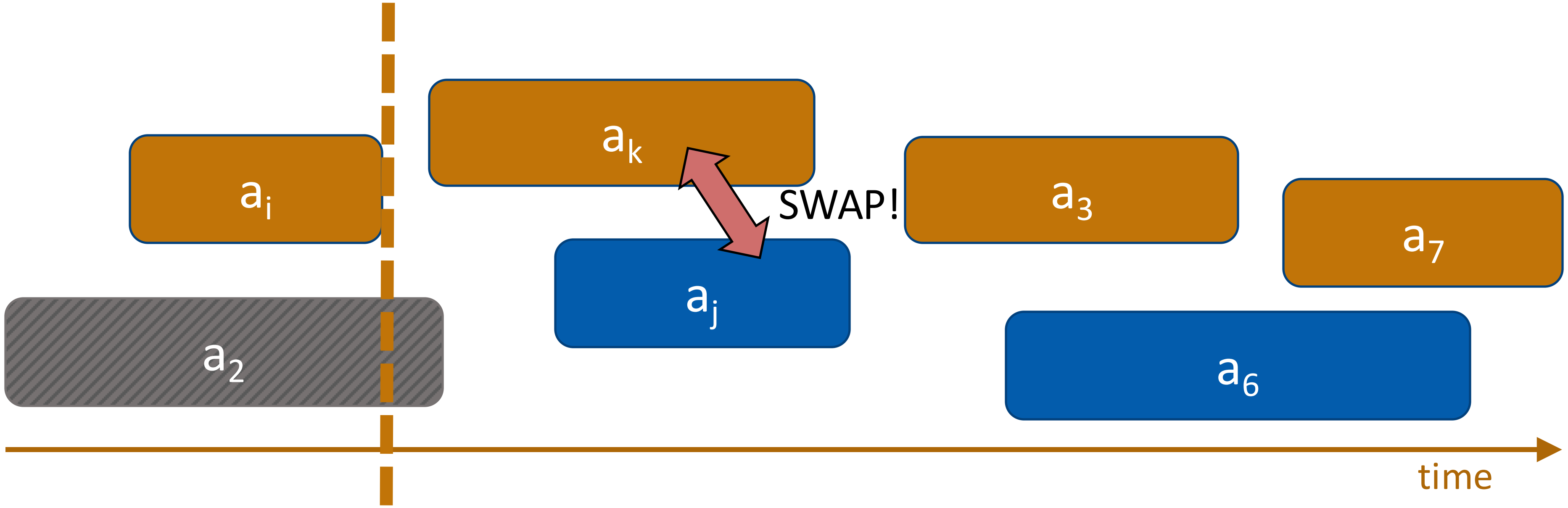
The Correctness of Activity Selection

- This schedule T is still allowed.
  - Since a_k has the smallest ending time, it ends before a_j.
  - Thus, a_k doesn't conflict with anything chosen after a_j.
- And T is still optimal.
  - It has the same number of activities as T*.



The Correctness of Activity Selection

- We've just shown:
  - If there was an optimal solution that extends the choices we made so far…
  - …then there is an optimal schedule that also contains our next greedy choice a_k



The Correctness of Activity Selection

So it's correct!

- **We never rule out an optimal solution**
- At the end of the algorithm, we've got some solution.
- So it must be optimal.

A Common Strategy

A common strategy for proving the correctness of greedy algorithms:

- Make a **series of choices**.
- Show that, at each step, our choice **won't rule out an optimal solution** at the end of the day.
- After we've made all our choices, we haven't ruled out an optimal solution, **so we must have found one.**

A Common Strategy

- Inductive Hypothesis:
  - After greedy choice t, you haven’t ruled out success.
- Base case:
  - Success is possible before you make any choices.
- Inductive step:
  - If you haven’t ruled out success after choice t, then you won’t rule out success after choice t+1.
- Conclusion:
  - If you reach the end of the algorithm and haven’t ruled out success then you must have succeeded.

A Common Strategy

A common strategy for showing we don’t rule out the optimal solution:

- Suppose that you’re on track to make an optimal solution T*.
  - E.g., after you’ve picked activity i, you’re still on track.
- Suppose that T* *disagrees* with your next greedy choice.
  - E.g., it *doesn’t* involve activity k.
- Manipulate T* in order to make a solution T that’s not worse but that *agrees* with your greedy choice.
  - E.g., swap whatever activity T* did pick next with activity k.

Three Questions

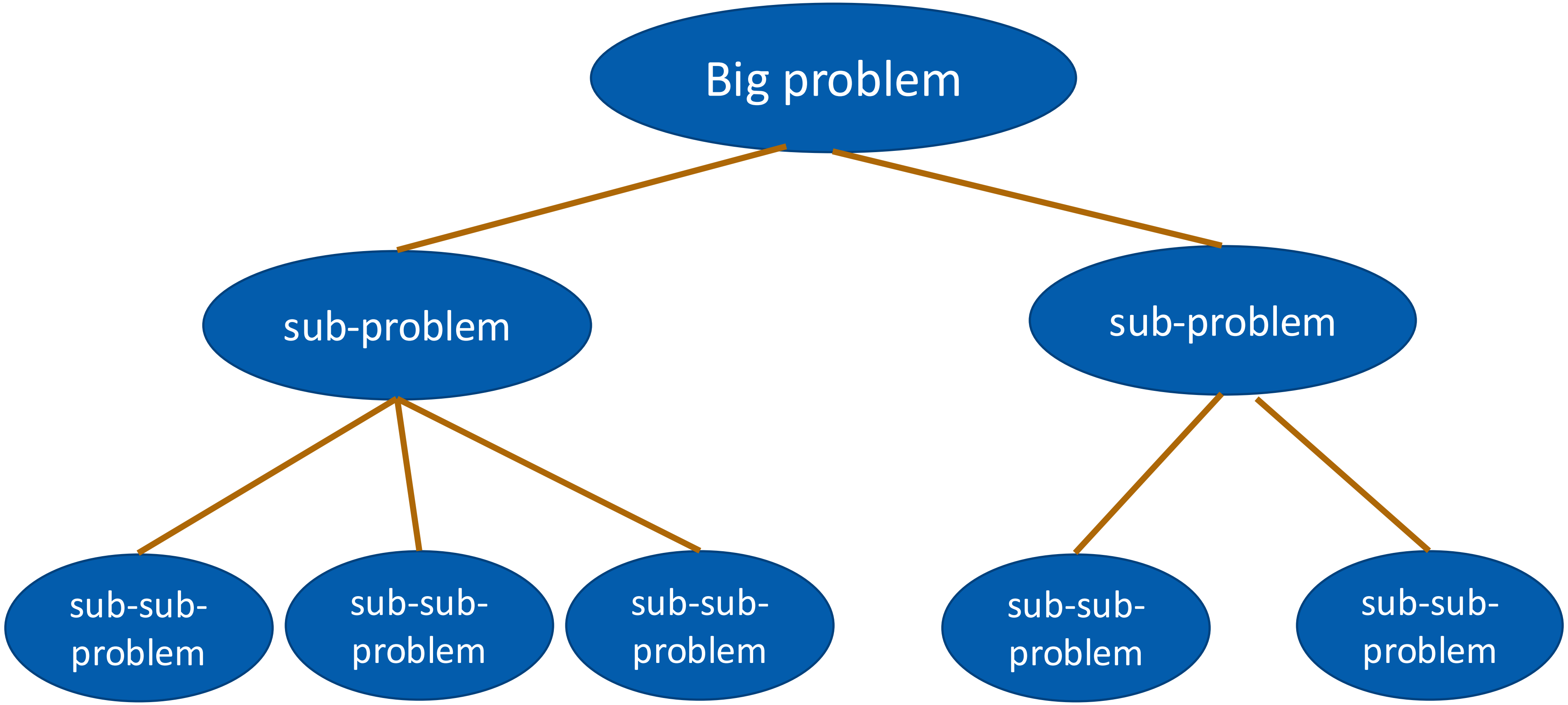
1. Does this greedy algorithm for activity selection work?
   – Yes

2. Greedy is simple. But why are we getting to it in week 11?
   – Proving that greedy algorithms work is often not so easy…

3. In general, when are greedy algorithms a good idea?
   – When the problem exhibits especially nice optimal substructure.

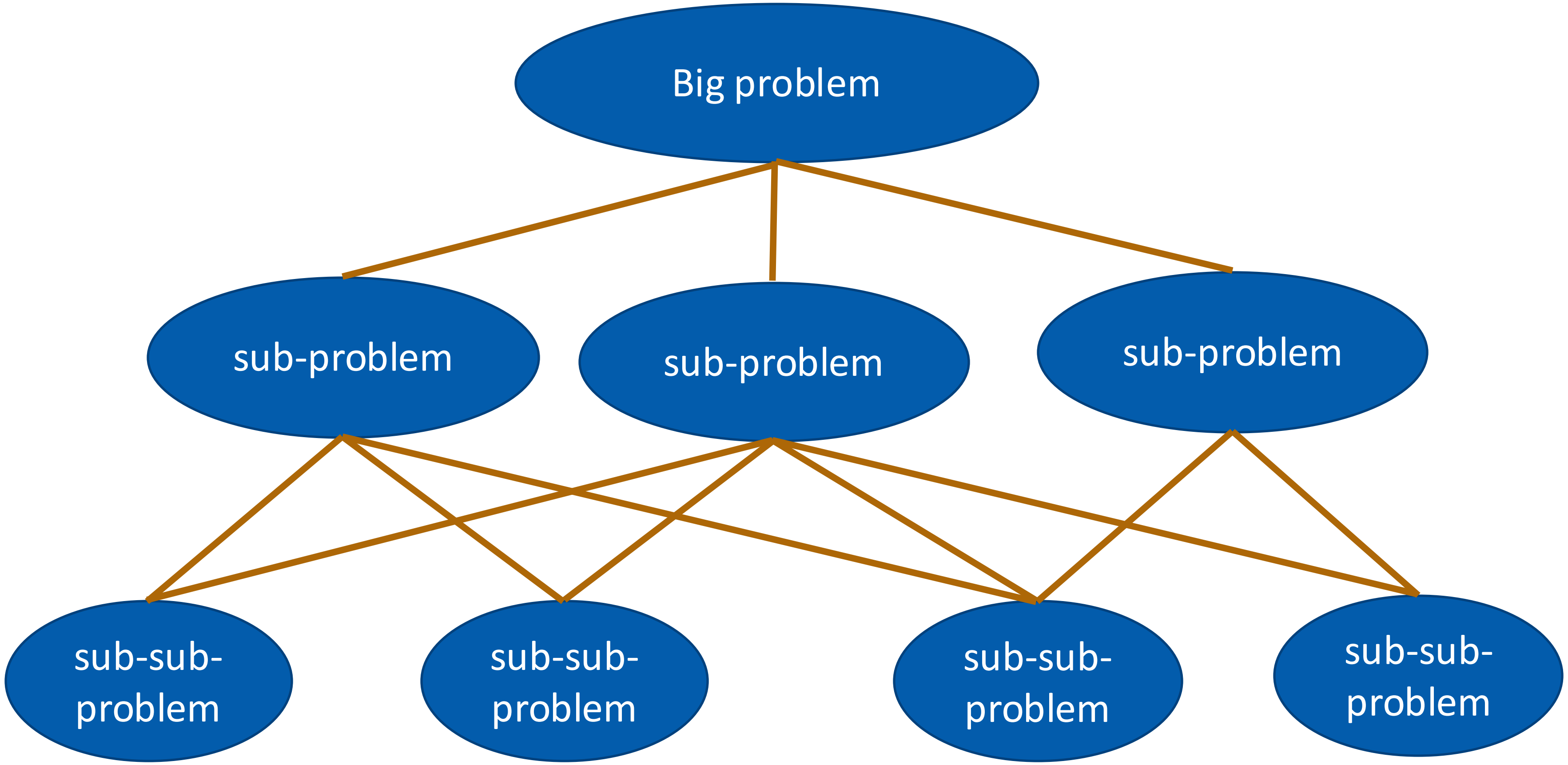
Sub-problem graph view

- Divide-and-conquer:



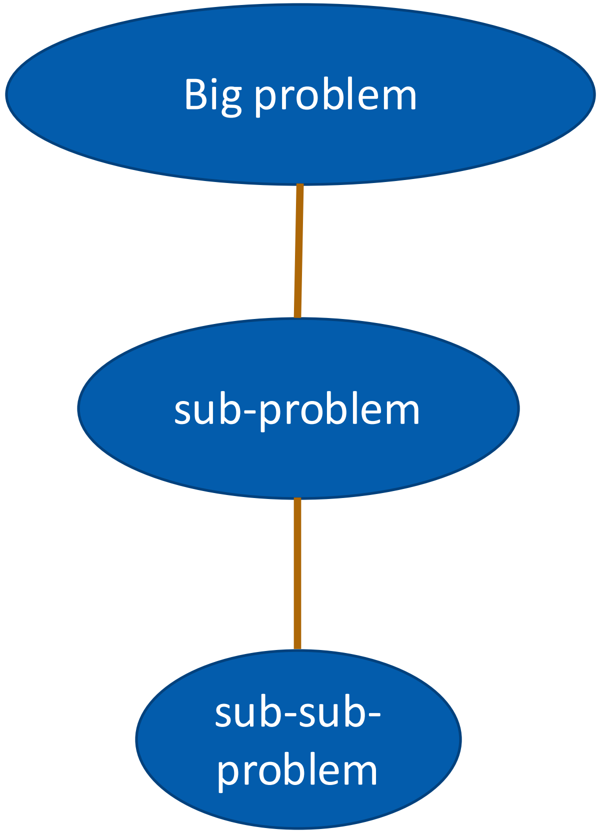
Sub-problem graph view

- Dynamic Programming:



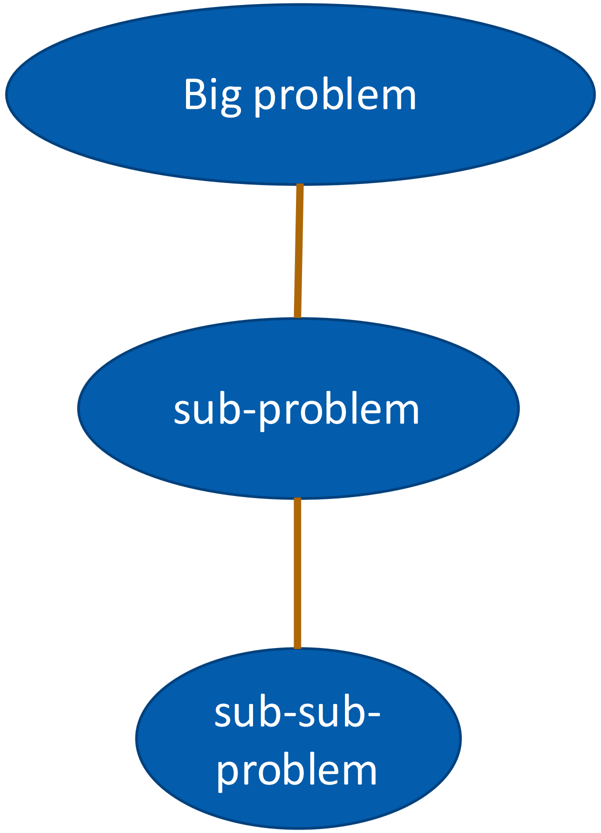
Sub-problem graph view

- Greedy algorithms:



Sub-problem graph view

- Greedy algorithms:



- Not only is there **optimal sub-structure:**
  - optimal solutions to a problem are made up from optimal solutions of sub-problems

- but each problem **depends on only one sub-problem.**

Three Questions

1. Does this greedy algorithm for activity selection work?
   – Yes

2. Greedy is simple. But why are we getting to it in week 11?
   – Proving that greedy algorithms work is often not so easy…

3. In general, when are greedy algorithms a good idea?
   – When the problem exhibits especially nice optimal substructure.

Another Example: Scheduling

- DSAA2043 HW
- Personal hygiene
- Math HW
- Administrative stuff for student club
- Econ HW
- Do laundry
- Sports
- Practice musical instrument
- Read lecture notes
- Have a social life
- Sleep

Scheduling

- n tasks
- Task i takes t_i hours
- For every hour that passes until task i is done, pay c_i

- DSAA2043 HW, then Sleep: costs $\mathbf{10 \cdot 2 + (10 + 8) \cdot 3 = 74}$ **units**
- Sleep, then DSAA2043 HW: costs $\mathbf{8 \cdot 3 + (10 + 8) \cdot 2 = 60}$ **units**

Scheduling

- This problem breaks up nicely into sub-problems:

Suppose this is the optimal schedule:

Then this must be the optimal schedule on just jobs B,C,D.

If not, then rearranging B,C,D could make a better schedule than (A,B,C,D)!

Scheduling

- Seems amenable to a greedy algorithm:

Take the best job first

Then solve this problem

Job A | Job B | Job C | Job D

Take the best job first

Then solve this problem

Job C | Job B | Job D

Take the best job first

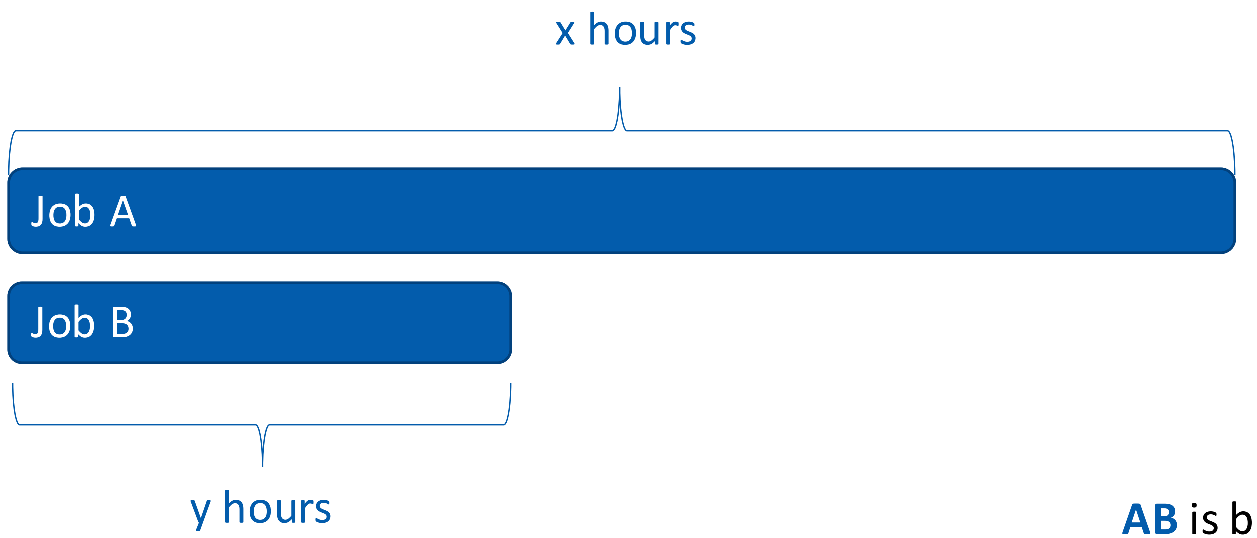
Then solve this problem

Job D | Job B

(That one's easy ☺)

What does “best” mean?

- Of these two jobs, which should we do first?



Cost: z units per hour until it’s done.

Cost: w units per hour until it’s done.

- Cost(**A then B**) = x · z + (x + y) · w
- Cost(**B then A**) = y · w + (x + y) · z

AB is better than **BA** when:

$$xz + (x + y)w \leq yw + (x + y)z$$
$$xz + xw + yw \leq yw + xz + yz$$
$$wx \leq yz$$
$$\frac{w}{y} \leq \frac{z}{x}$$

Idea for Greedy

- Choose the job with the biggest $\frac{\text{cost of delay}}{\text{time it takes}}$ ratio.

Correctness

- Suppose you have already chosen some jobs, and haven't yet ruled out success:

Job E	Job C	Job A	Job B	Job D

- Then if you choose the next job to be the one left that maximizes the ratio **cost/time**, you still won't rule out success.
- **Proof sketch:**
  - Say Job B maximizes this ratio, but it's not the next job in the opt. soln.
  - Switch A and B! Nothing else will change, and we just showed that the cost of the solution won't increase.

Job E	Job C	Job B	Job A	Job D

  - Repeat until B is first.

Job E	Job B	Job C	Job A	Job D

  - Now this is an optimal schedule where B is first.

Correctness

- Inductive Hypothesis:
  - After greedy choice t, you haven't ruled out success.
- Base case:
  - Success is possible before you make any choices.
- Inductive step:
  - If you haven't ruled out success after choice t, then you won't rule out success after choice t+1.
- Conclusion:
  - If you reach the end of the algorithm and haven't ruled out success then you must have succeeded.

Greedy Scheduling Solution

- **scheduleJobs**(JOBS):
  - Sort JOBS in decreasing order by the ratio:
    - $r_i = \frac{c_i}{t_i} = \frac{\text{cost of delaying job i}}{\text{time job i takes to complete}}$
  - **Return** JOBS

Running time: O(n log(n))

Minimum Spanning Trees

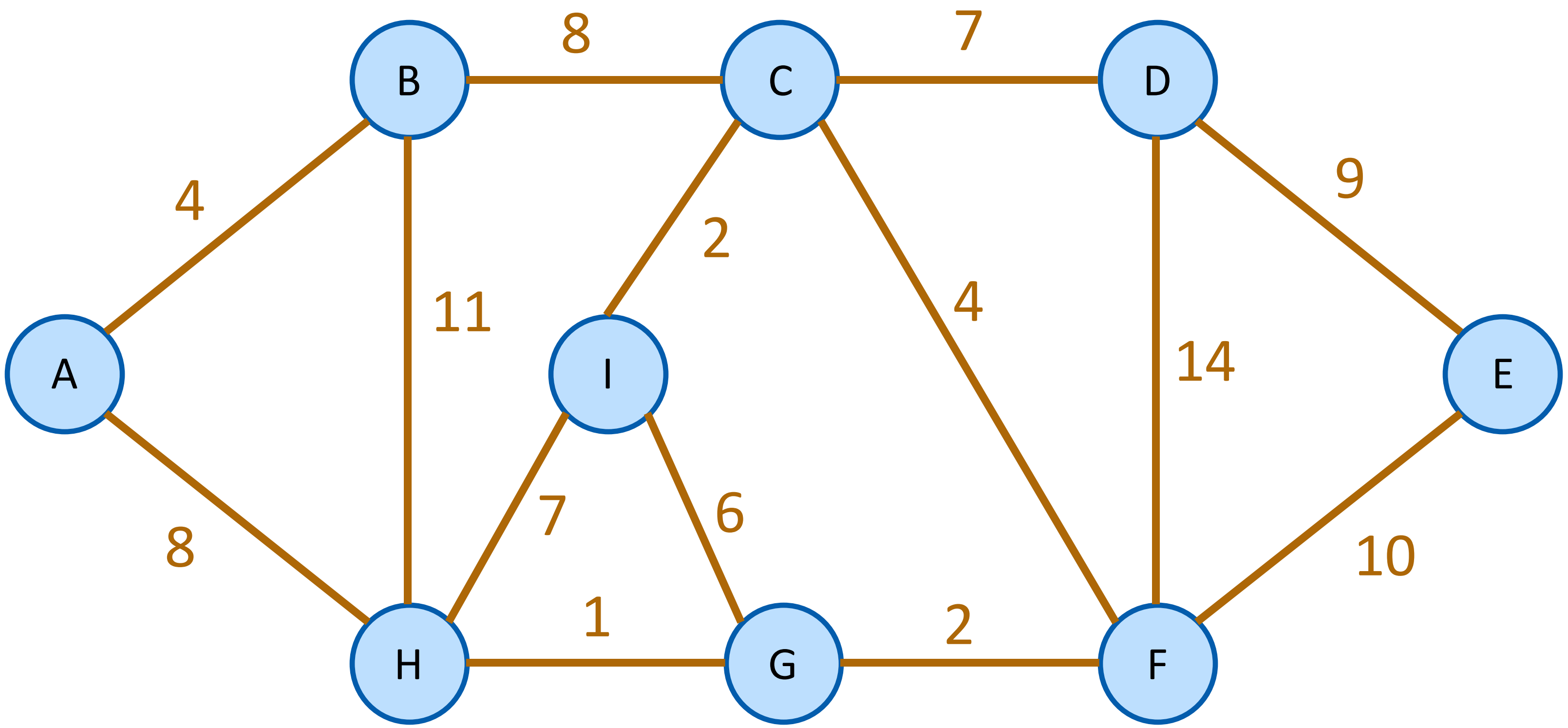
Minimum Spanning Trees

- 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 Trees

- Say we have an undirected weighted graph



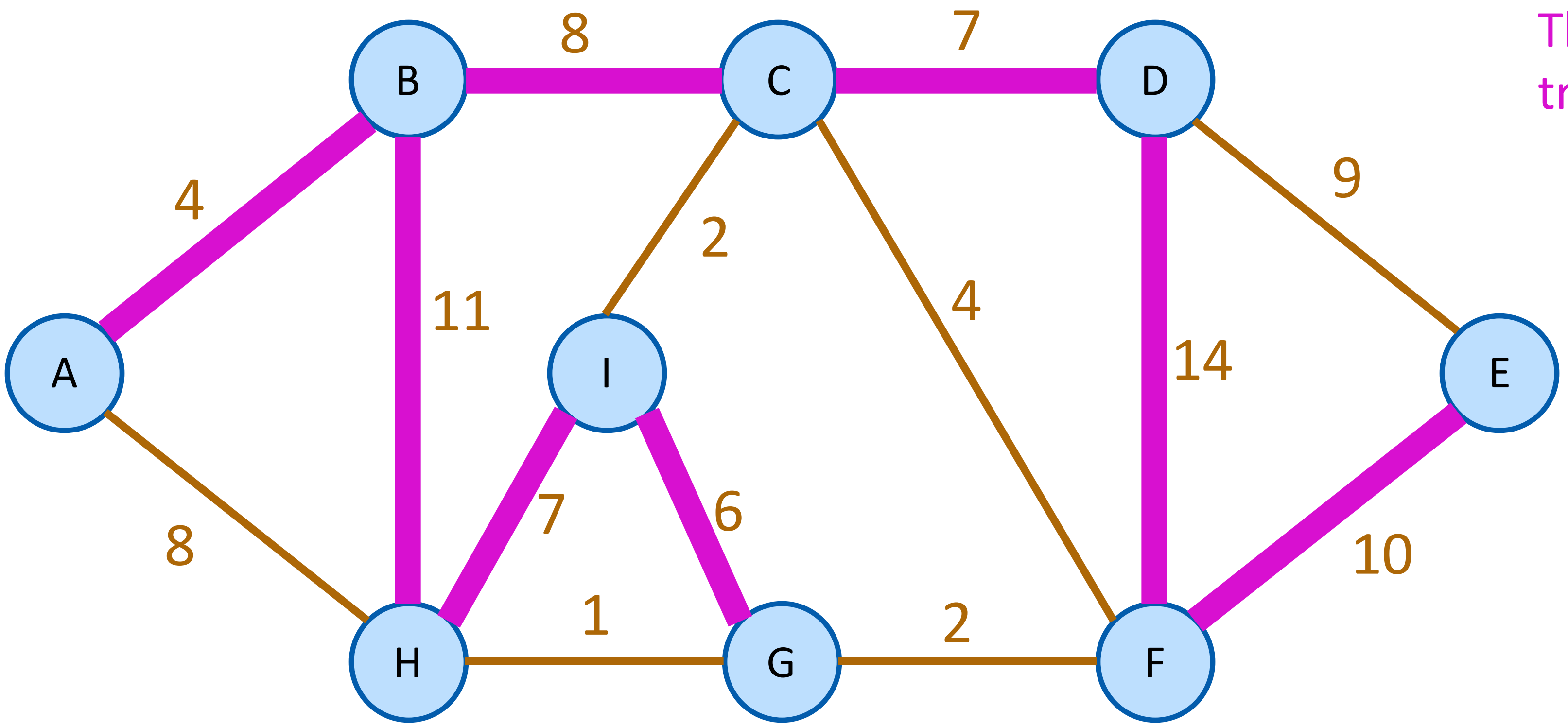
A **tree** is a connected graph with no cycles!

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

Minimum Spanning Trees

- Say we have an undirected weighted graph

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

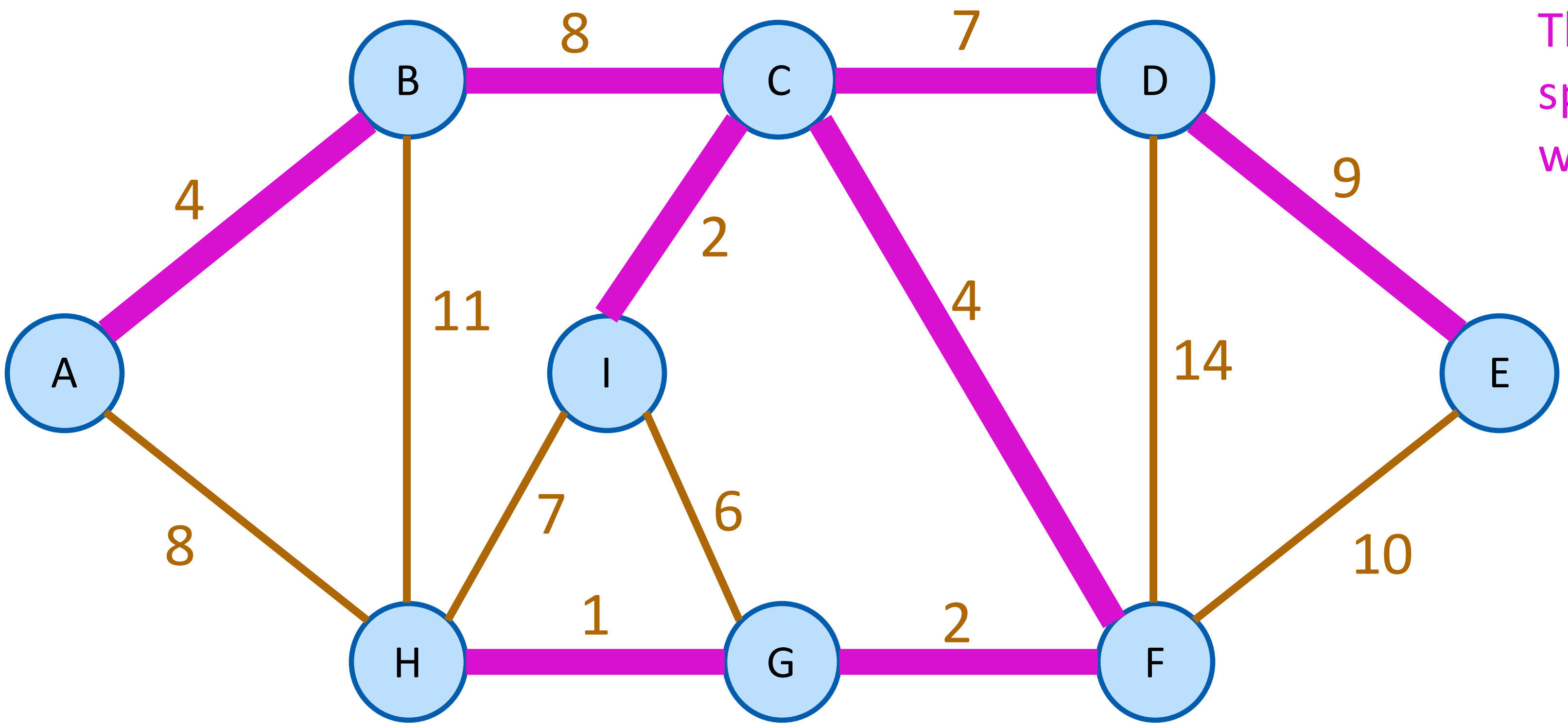


This is a spanning tree with cost 67.

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

Minimum Spanning Trees

- Say we have an undirected weighted graph

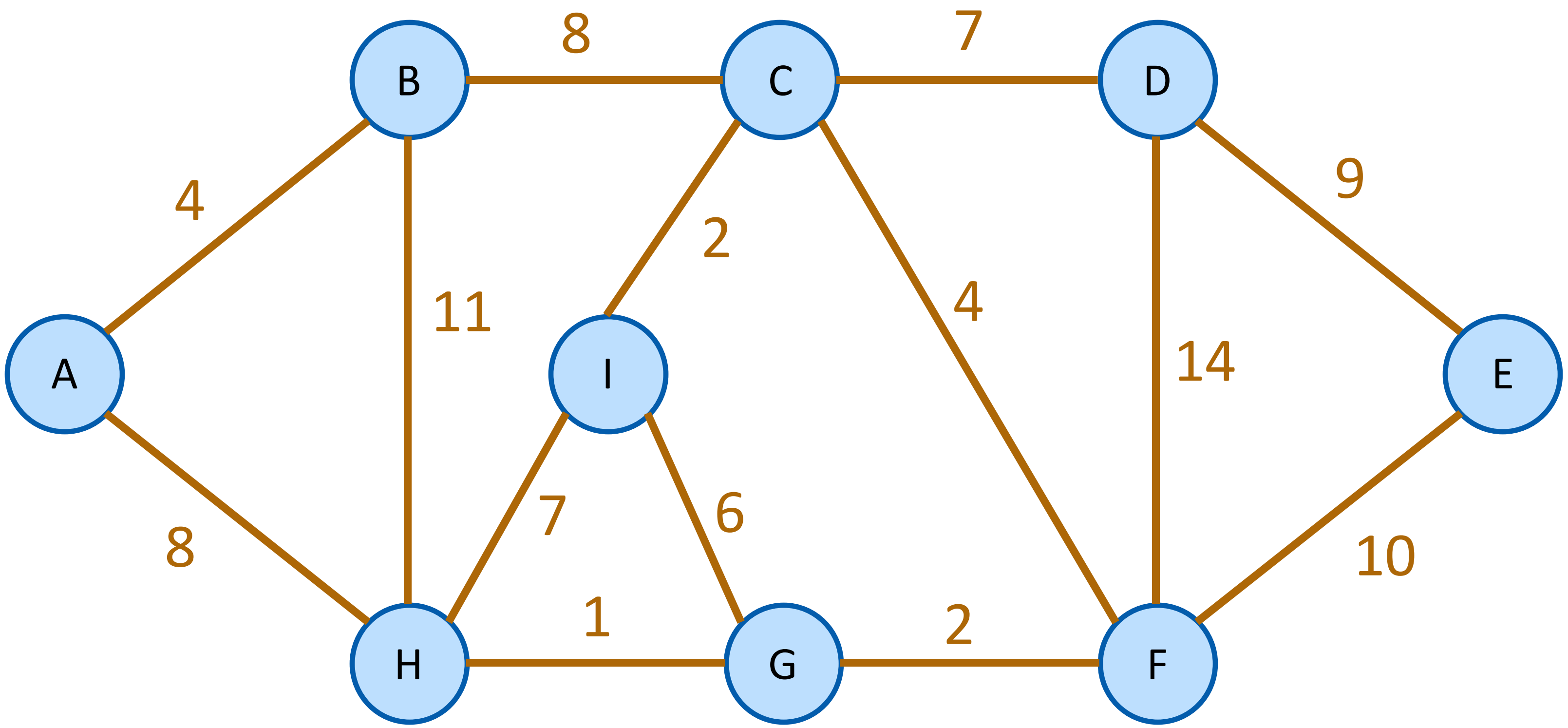


This is also a spanning tree, with cost 37.

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

Minimum Spanning Trees

- Say we have an undirected weighted graph



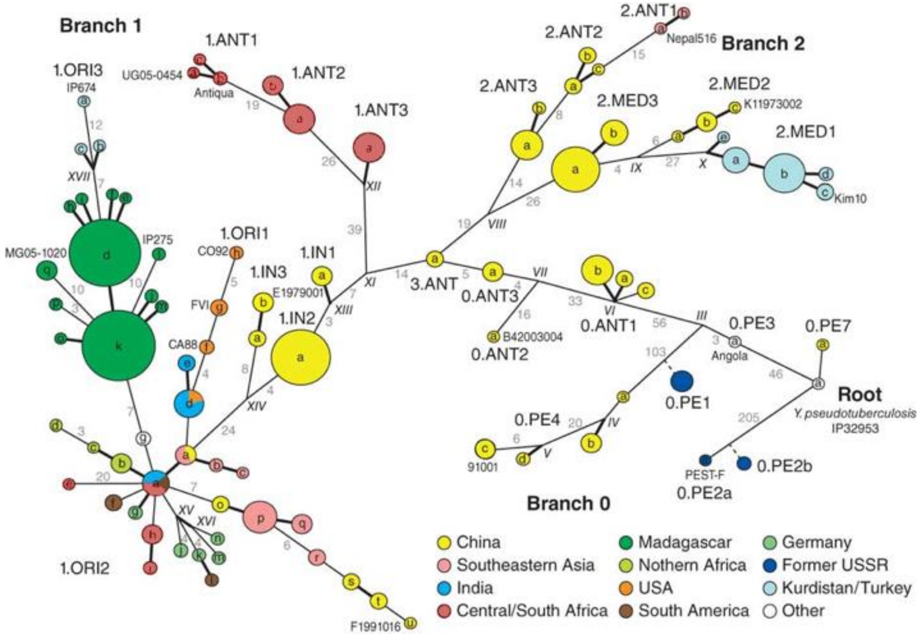
minimum

of minimum 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
  - E.g., genetic distance
- Image processing
  - E.g., image segmentation
- Useful primitive
  - For other graph algs



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.

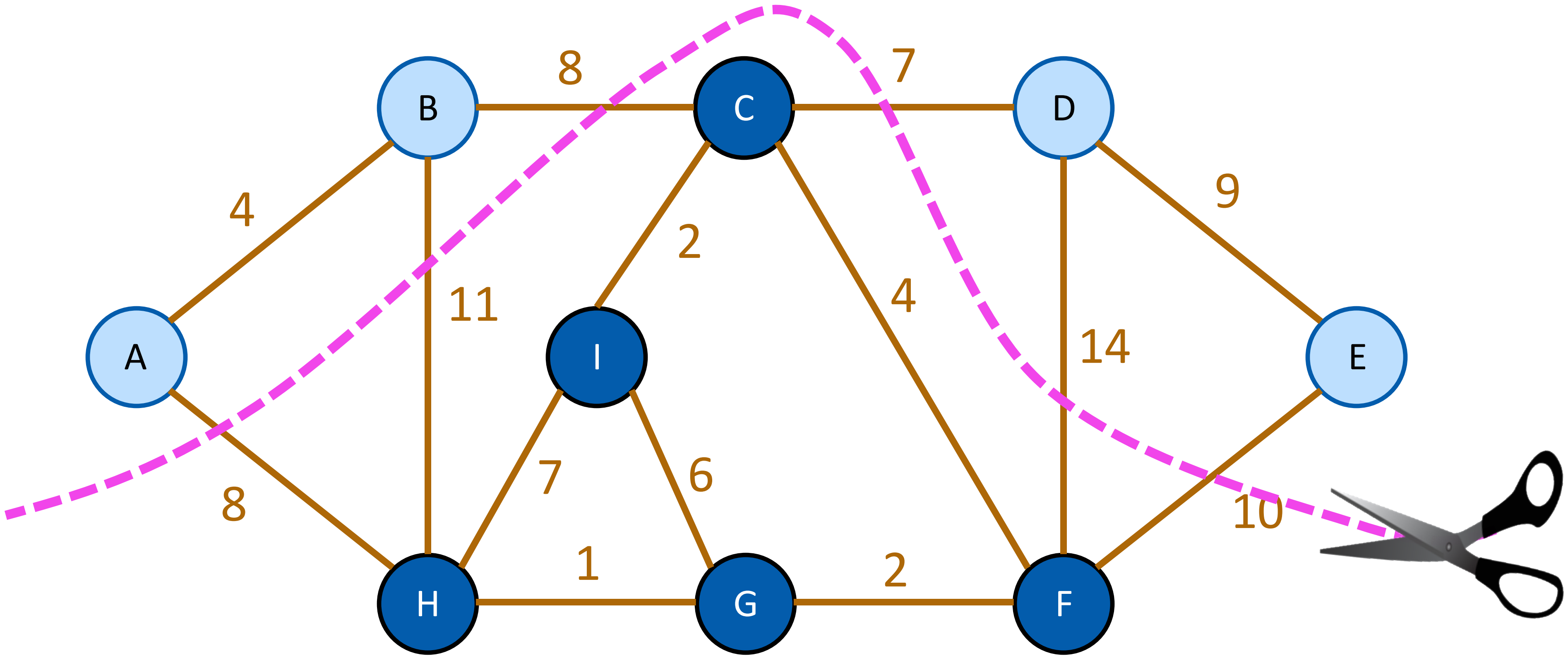
Then the next greedy choice that we make

is still consistent with an MST.

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

Brief Aside – Cuts in Graphs

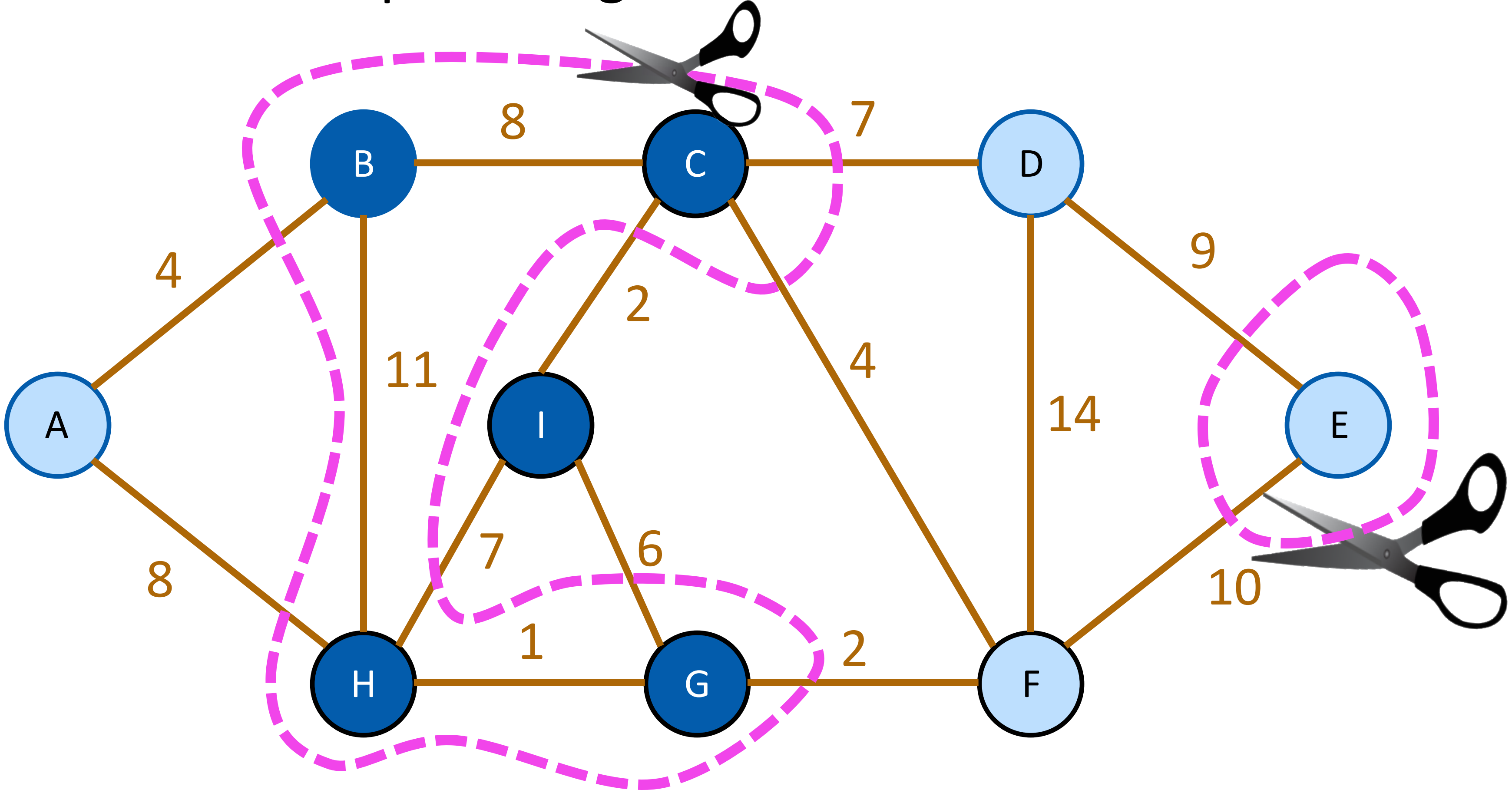
- A cut is a partition of the vertices into two parts:



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

Brief Aside – Cuts in Graphs

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

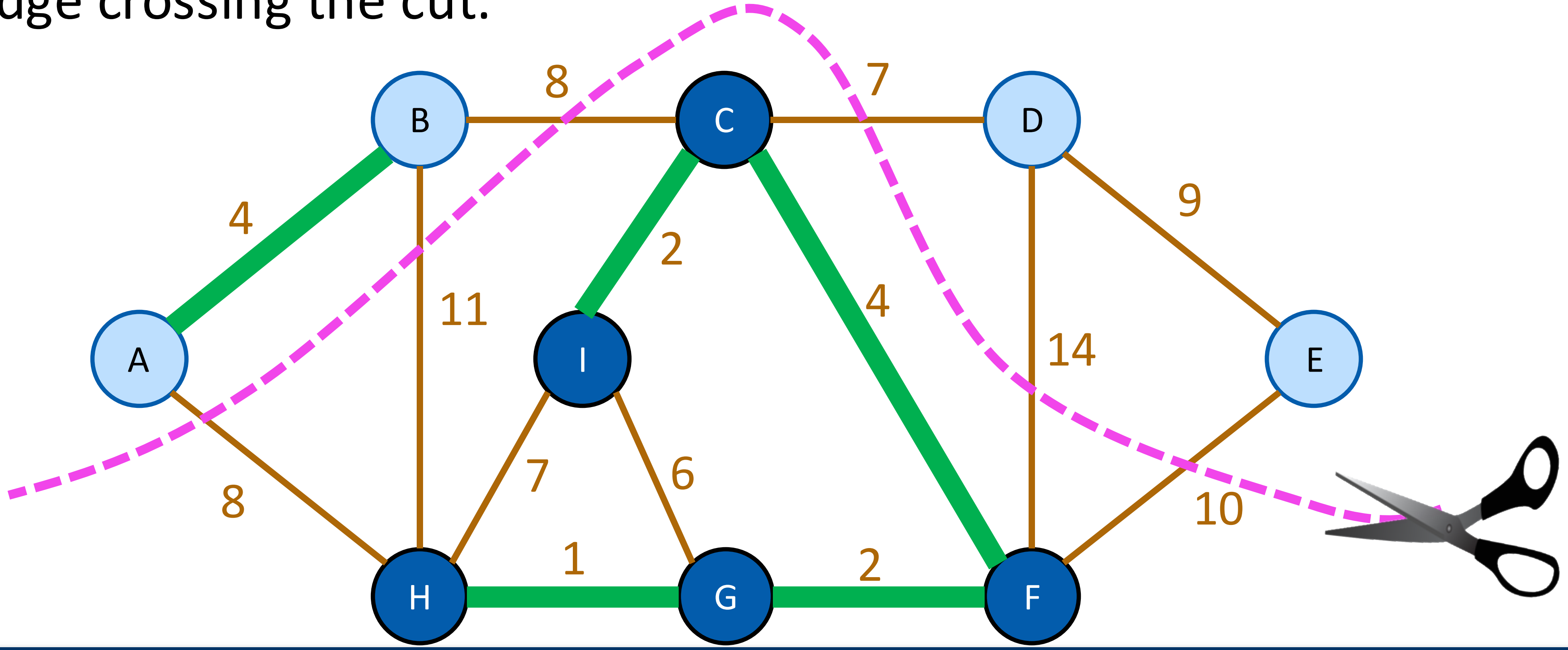


This is the cut **“{B,C,E,G,H} and {A,D,I,F}”**

Brief Aside – Cuts in Graphs

Let S be a set of edges in G

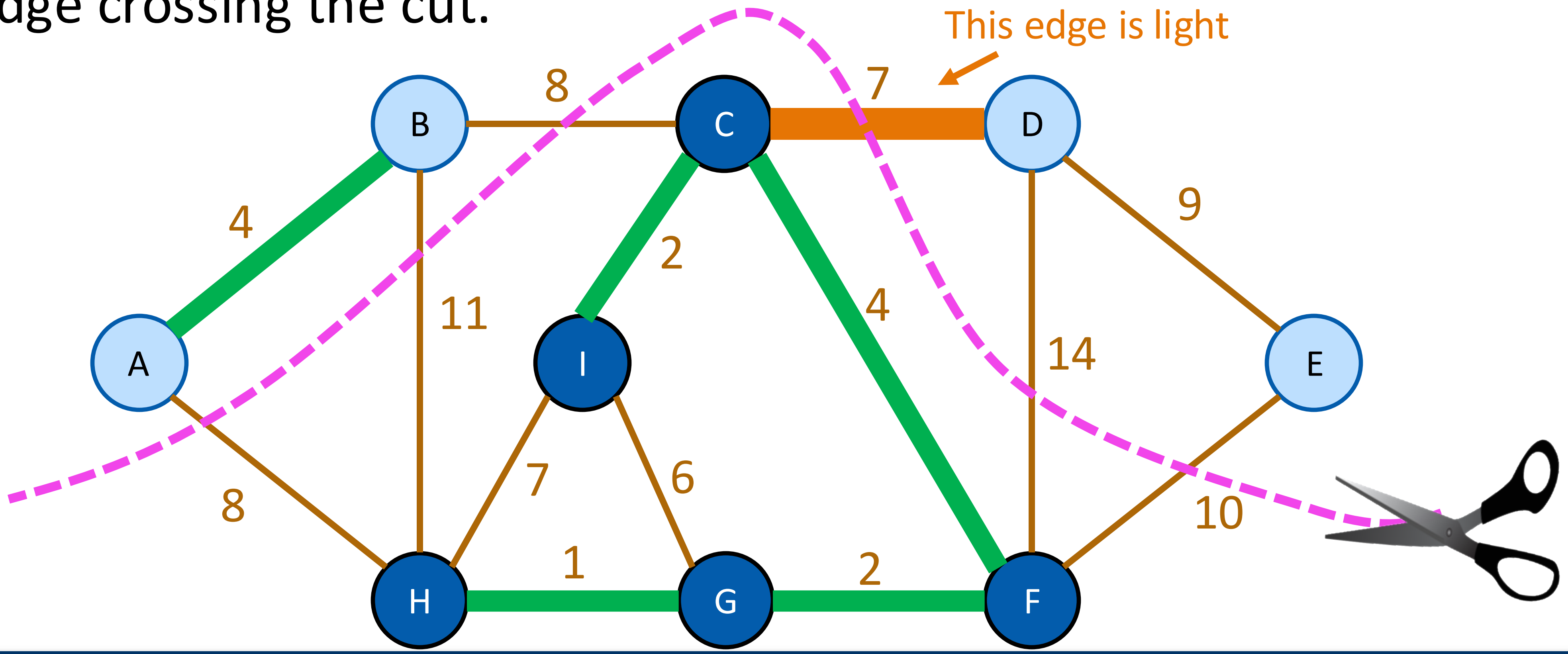
- We say a cut **respects** S if no edges in S cross the cut.
- An edge crossing a cut is called **light** if it has the smallest weight of any edge crossing the cut.



Brief Aside – Cuts in Graphs

Let S be a set of edges in G

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

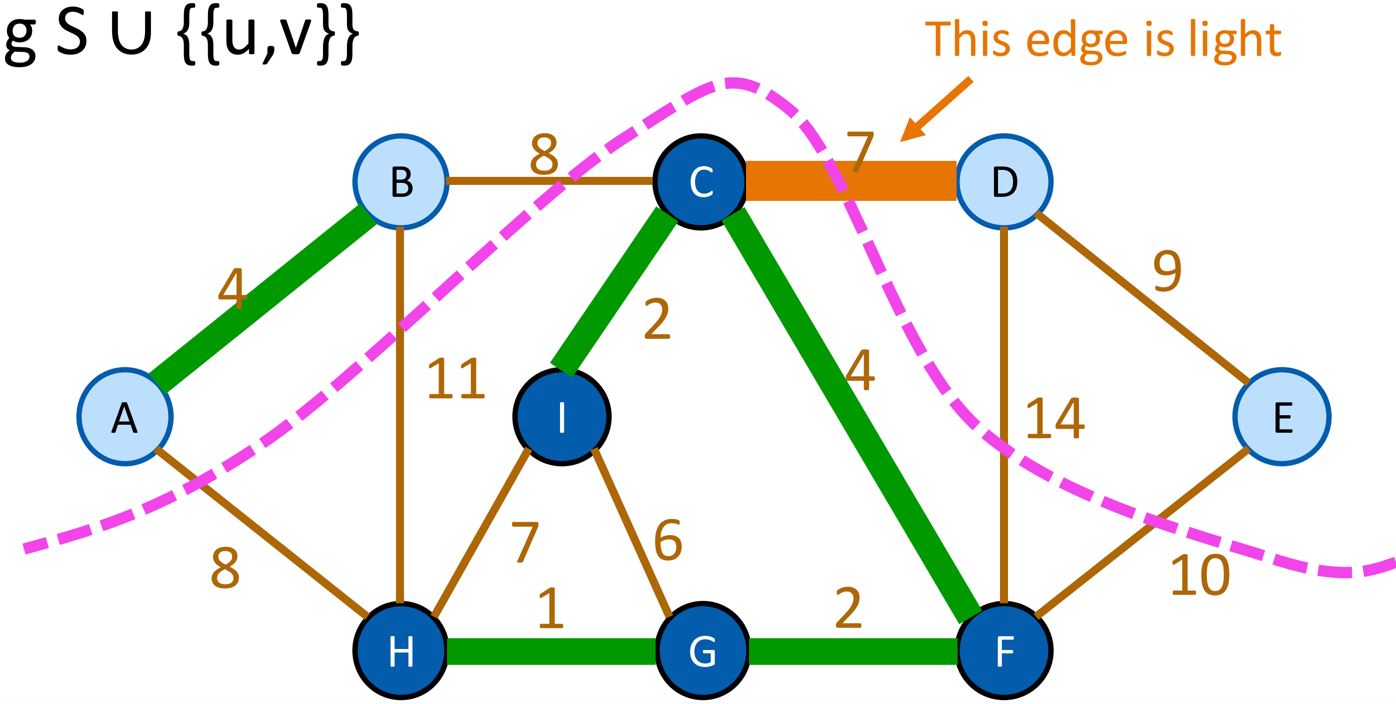


Lemma

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

Aka:

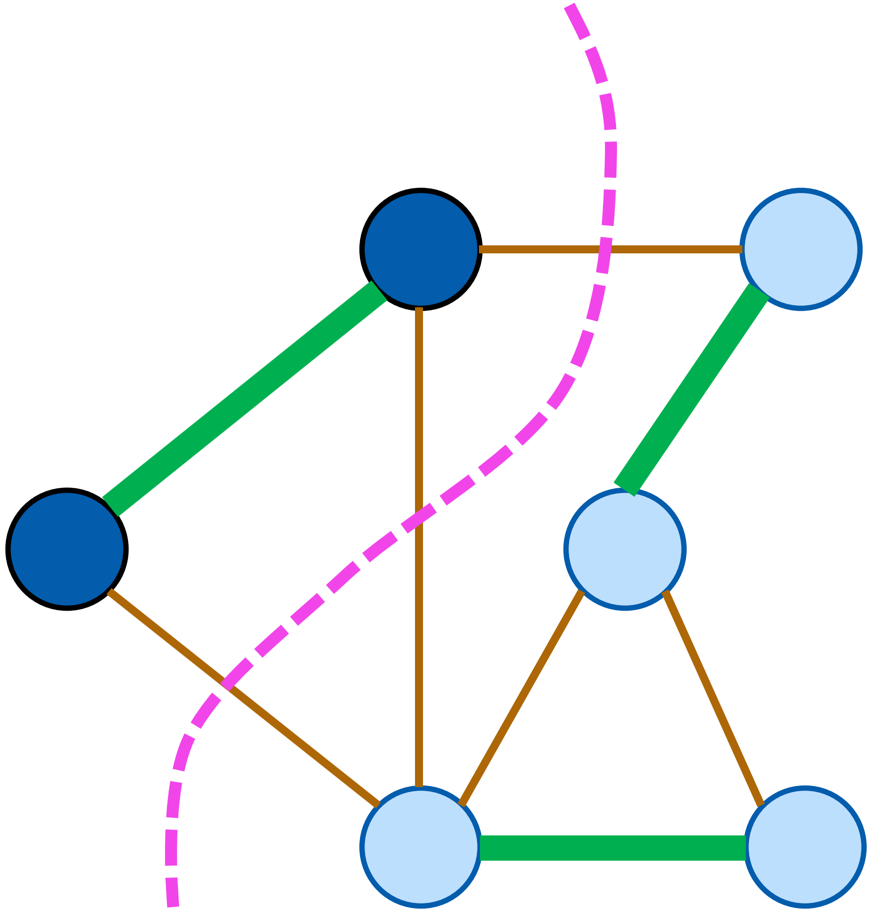
If we haven't ruled out the possibility of success so far, then adding a light edge still won't rule it out.



Brief Aside – Cuts in Graphs

Proof of Lemma

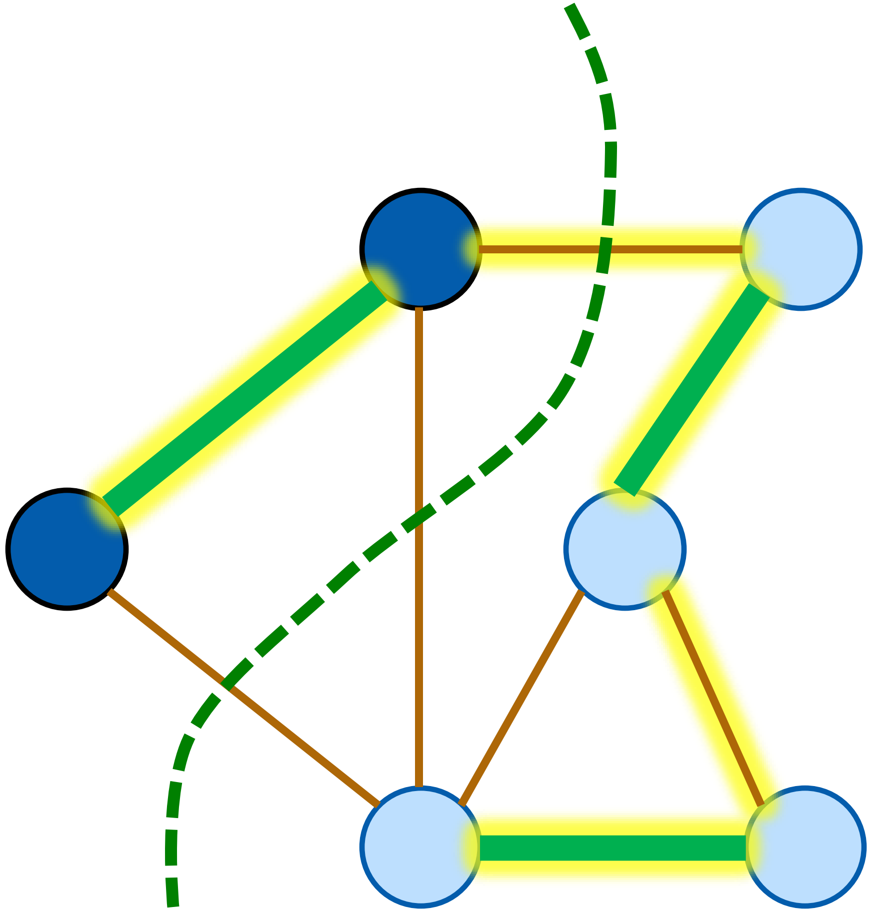
- Assume that we have:
  - a **cut** that respects **S**



Brief Aside – Cuts in Graphs

Proof of Lemma

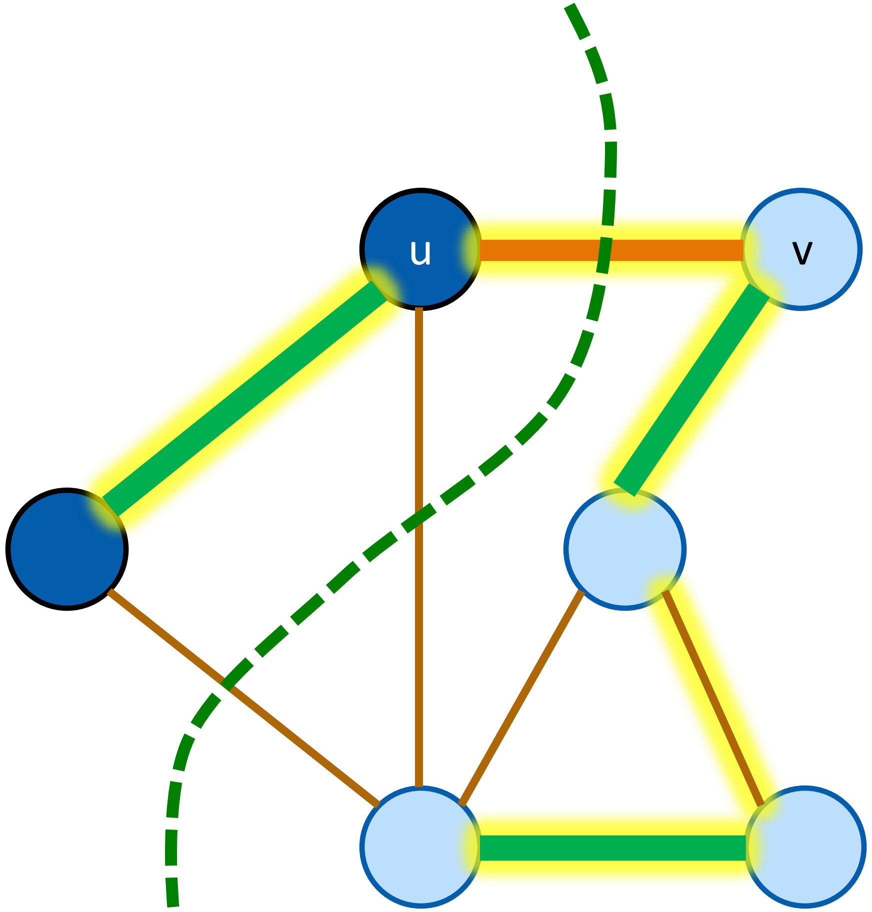
- Assume that we have:
  - a **cut** that respects **S**
  - **S** is part of some MST T.



Brief Aside – Cuts in Graphs

Proof of Lemma

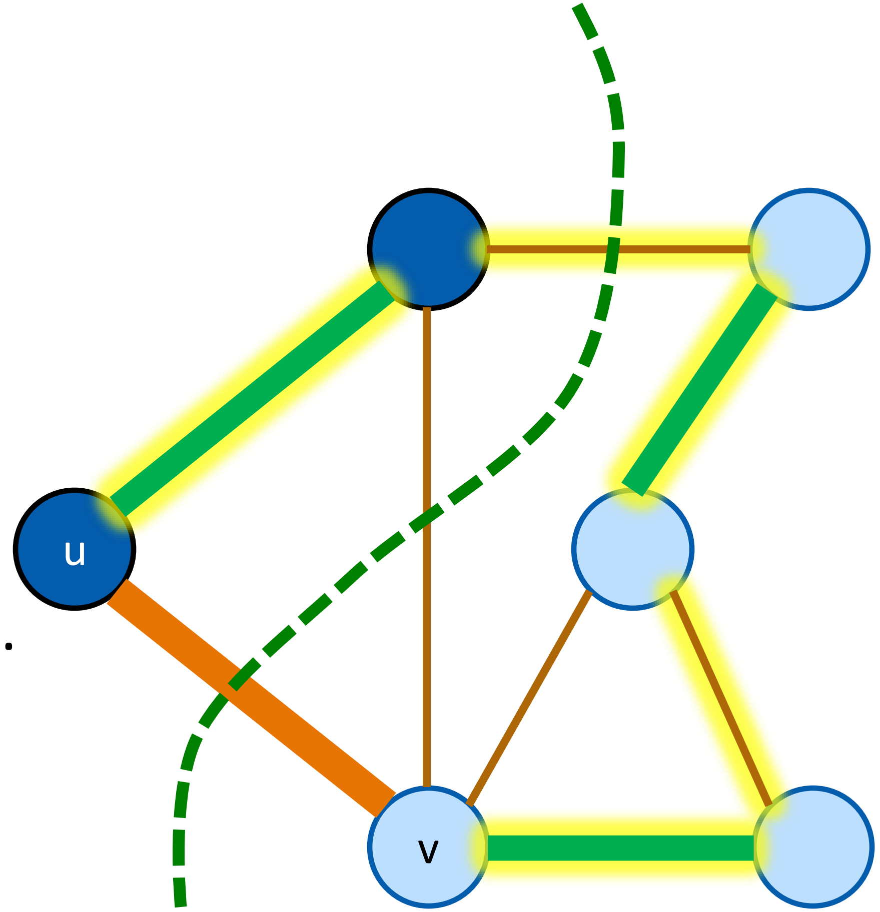
- Assume that we have:
  - a **cut** that respects **S**
  - **S** is part of some MST T.
- Say that **{u,v}** is light.
  - lowest cost crossing the cut
- If **{u,v}** is in T, we are done.
  - T is an MST containing both {u,v} and S.



Brief Aside – Cuts in Graphs

Proof of Lemma

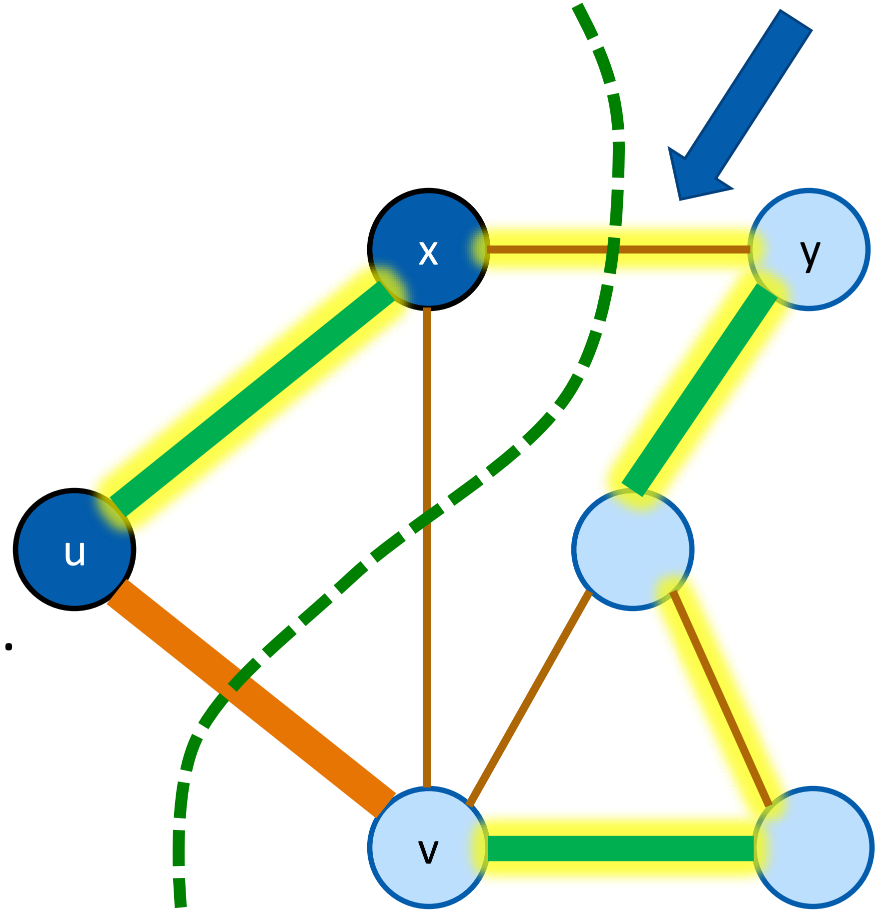
- Assume that we have:
  - a **cut** that respects **S**
  - **S** is part of some MST T.
- Say that **{u,v}** is light.
  - lowest cost crossing the cut
- Say **{u,v}** is not in **T**.
  - Note that adding **{u,v}** to **T** will make a cycle.



Brief Aside – Cuts in Graphs

Proof of Lemma

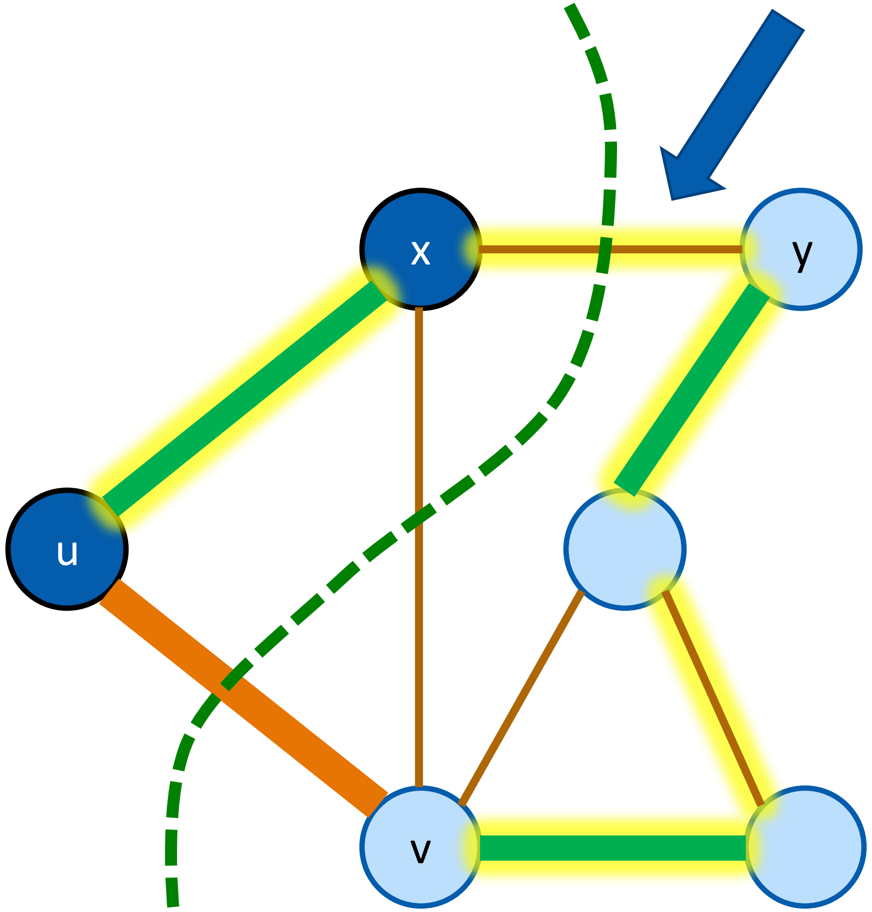
- Assume that we have:
  - a **cut** that respects **S**
  - **S** is part of some MST T.
- Say that **{u,v}** is light.
  - lowest cost crossing the cut
- Say **{u,v}** is not in **T**.
  - Note that adding **{u,v}** to **T** will make a cycle.
- There is at least one other edge, **{x,y}**, in this cycle crossing the cut.



Brief Aside – Cuts in Graphs

Proof of Lemma ctd.

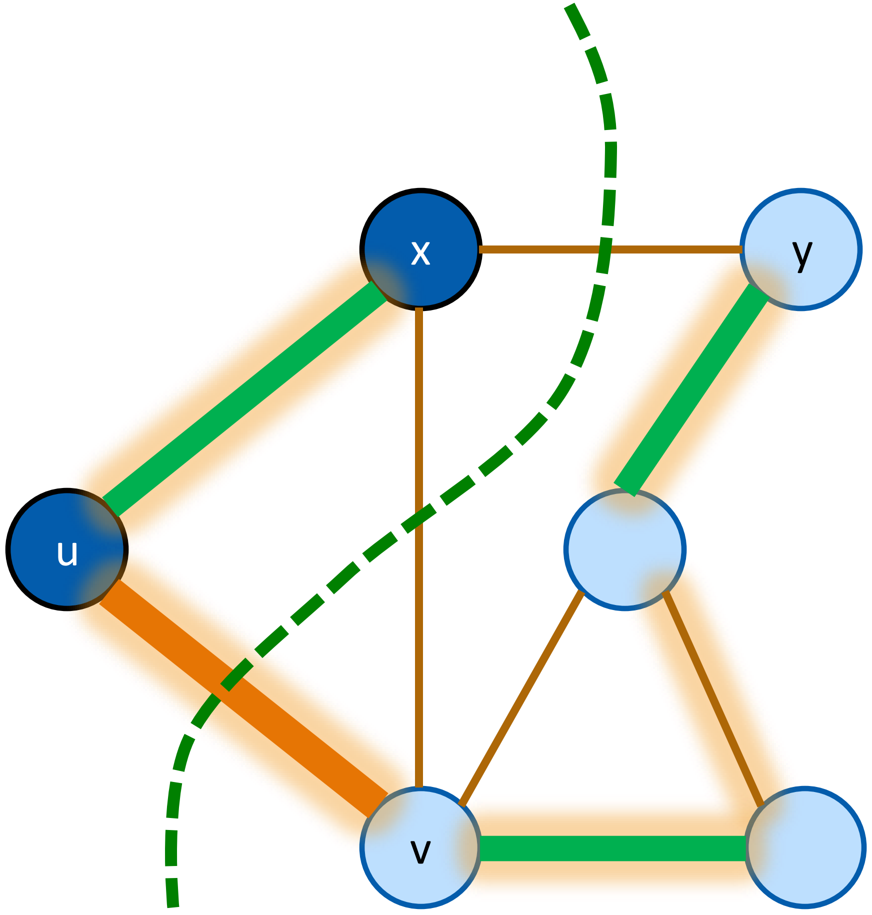
- Consider swapping {u,v} for {x,y} in **T**.
  - Call the resulting tree **T'.**



Brief Aside – Cuts in Graphs

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**
  - **T** had minimal cost.
  - So **T'** does too.
- So **T'** is an MST containing S and {u,v}.



How to find an MST

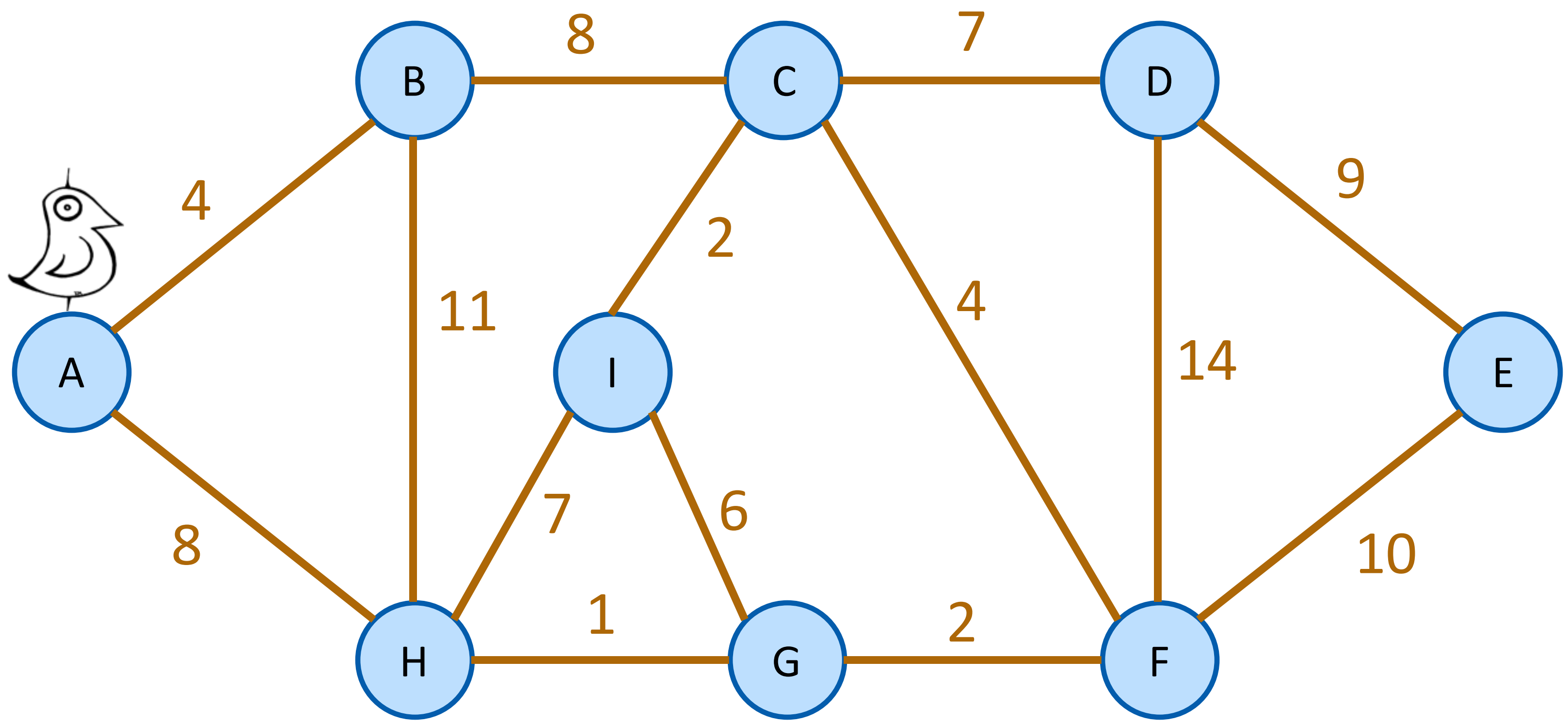
- How do we find one?
- Today we'll see **two greedy algorithms**.

- 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.

How to find an MST

Idea:

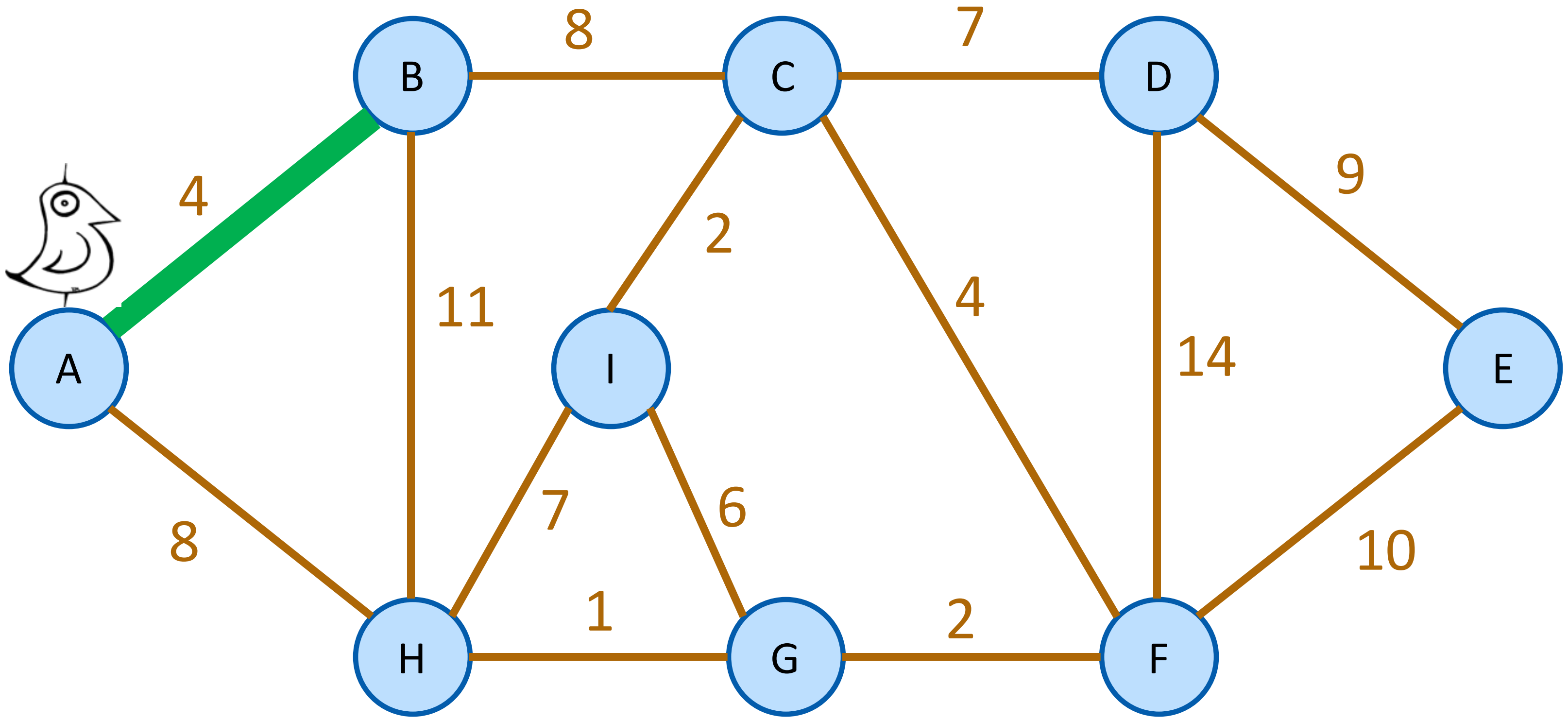
Start growing a tree, greedily add the shortest edge we can to grow the tree.



How to find an MST

Idea:

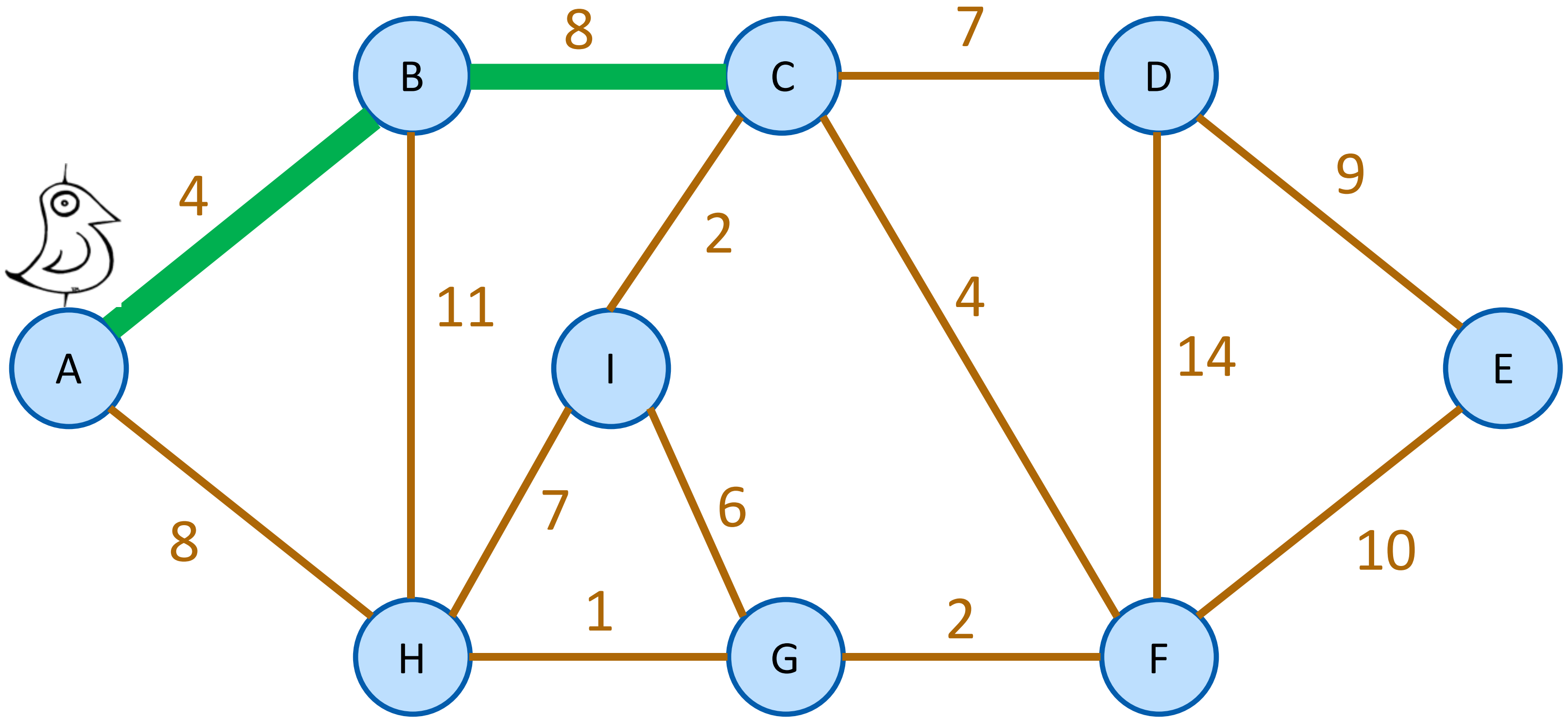
Start growing a tree, greedily add the shortest edge we can to grow the tree.



How to find an MST

Idea:

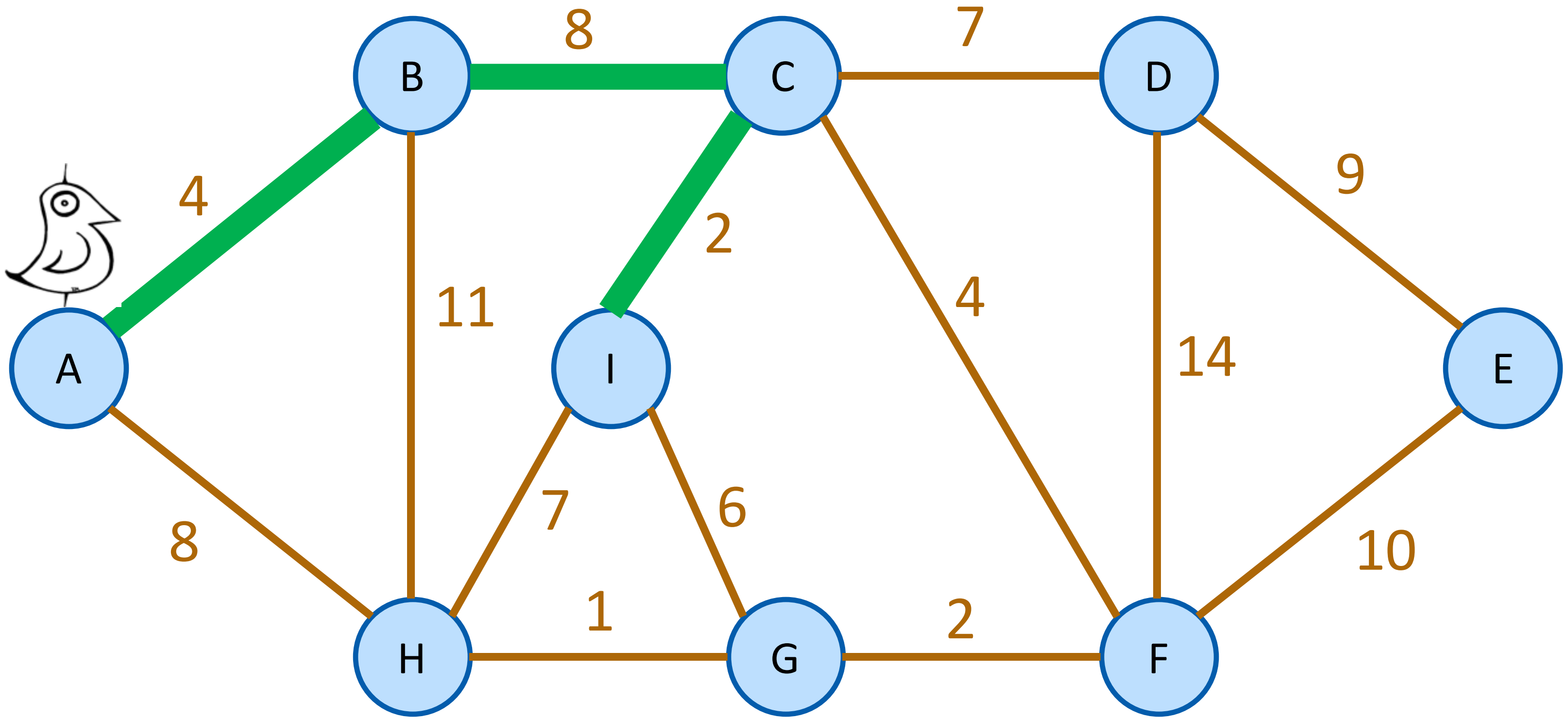
Start growing a tree, greedily add the shortest edge we can to grow the tree.



How to find an MST

Idea:

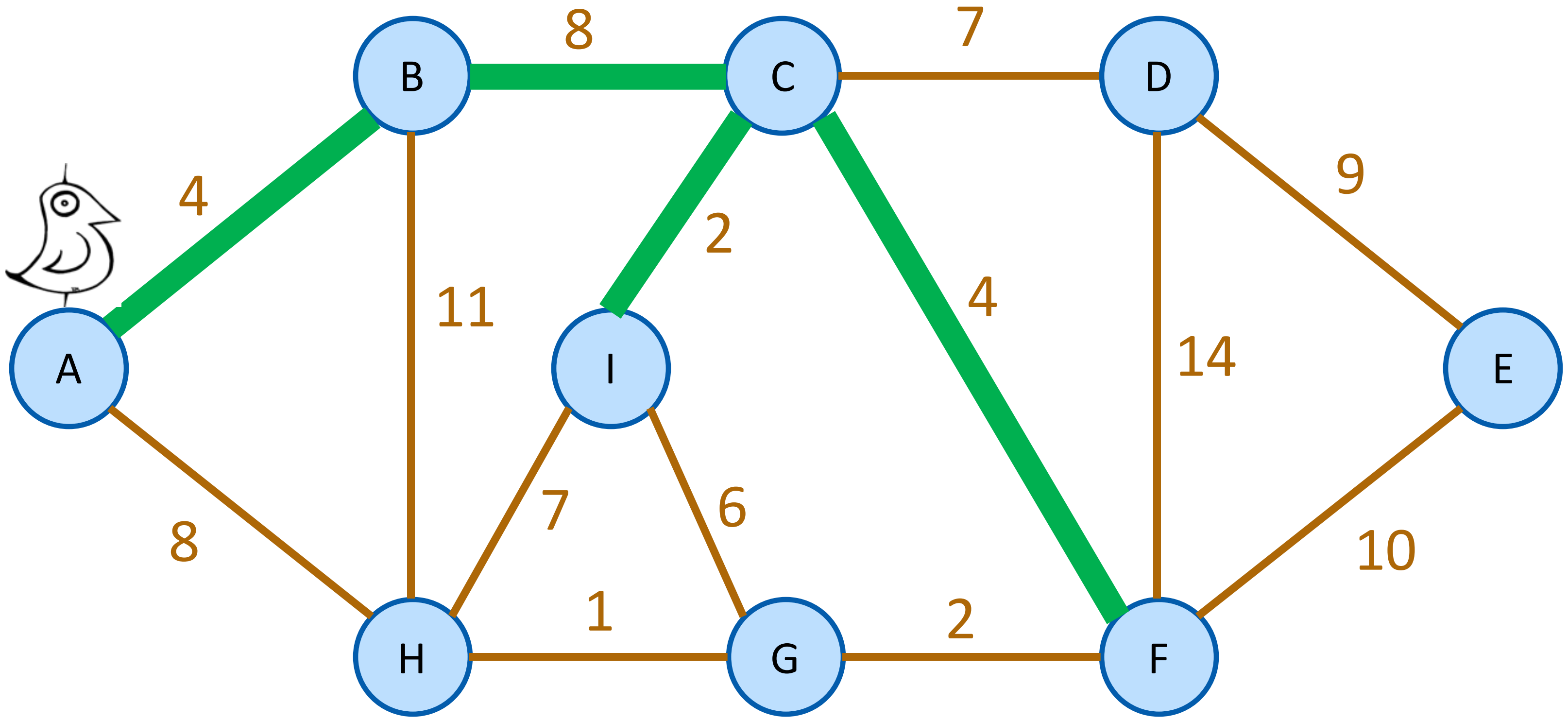
Start growing a tree, greedily add the shortest edge we can to grow the tree.



How to find an MST

Idea:

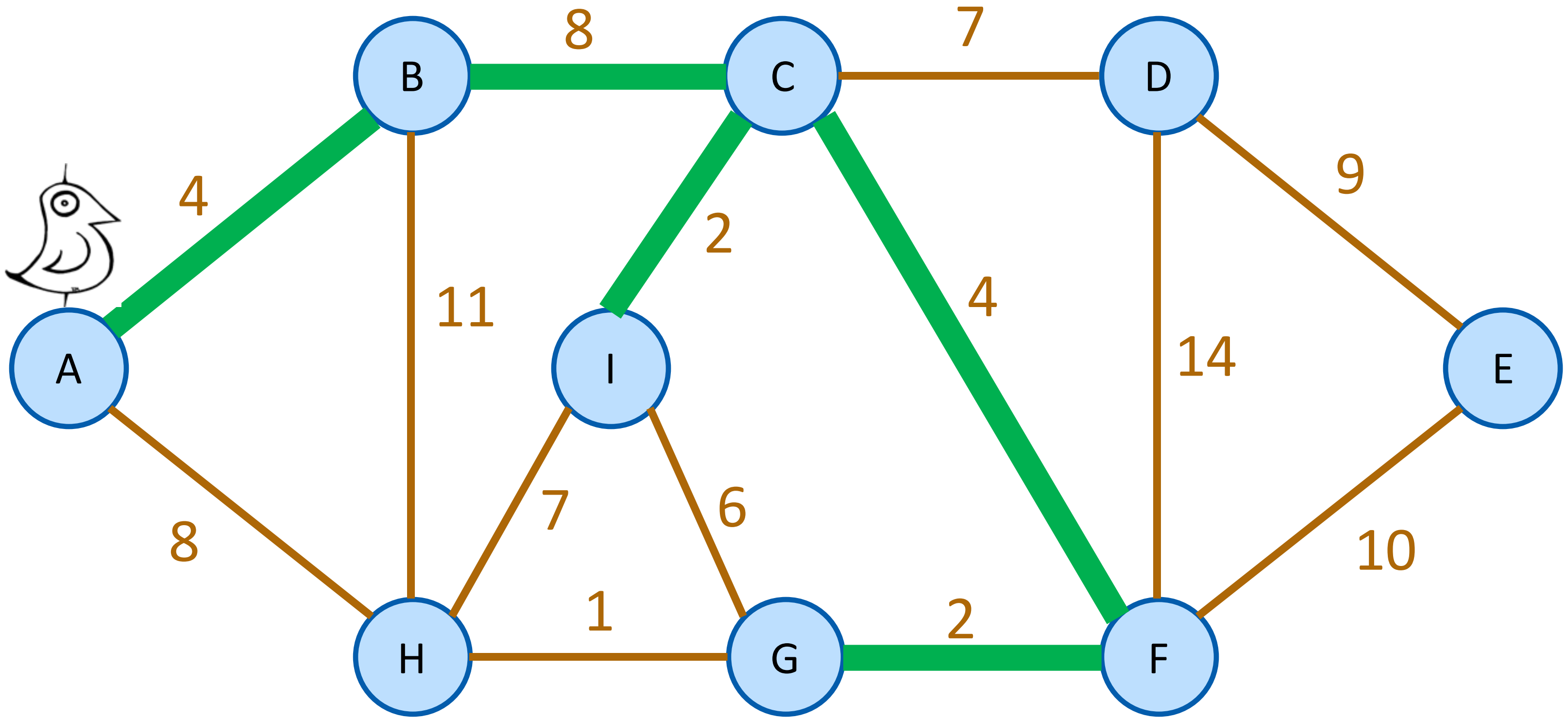
Start growing a tree, greedily add the shortest edge we can to grow the tree.



How to find an MST

Idea:

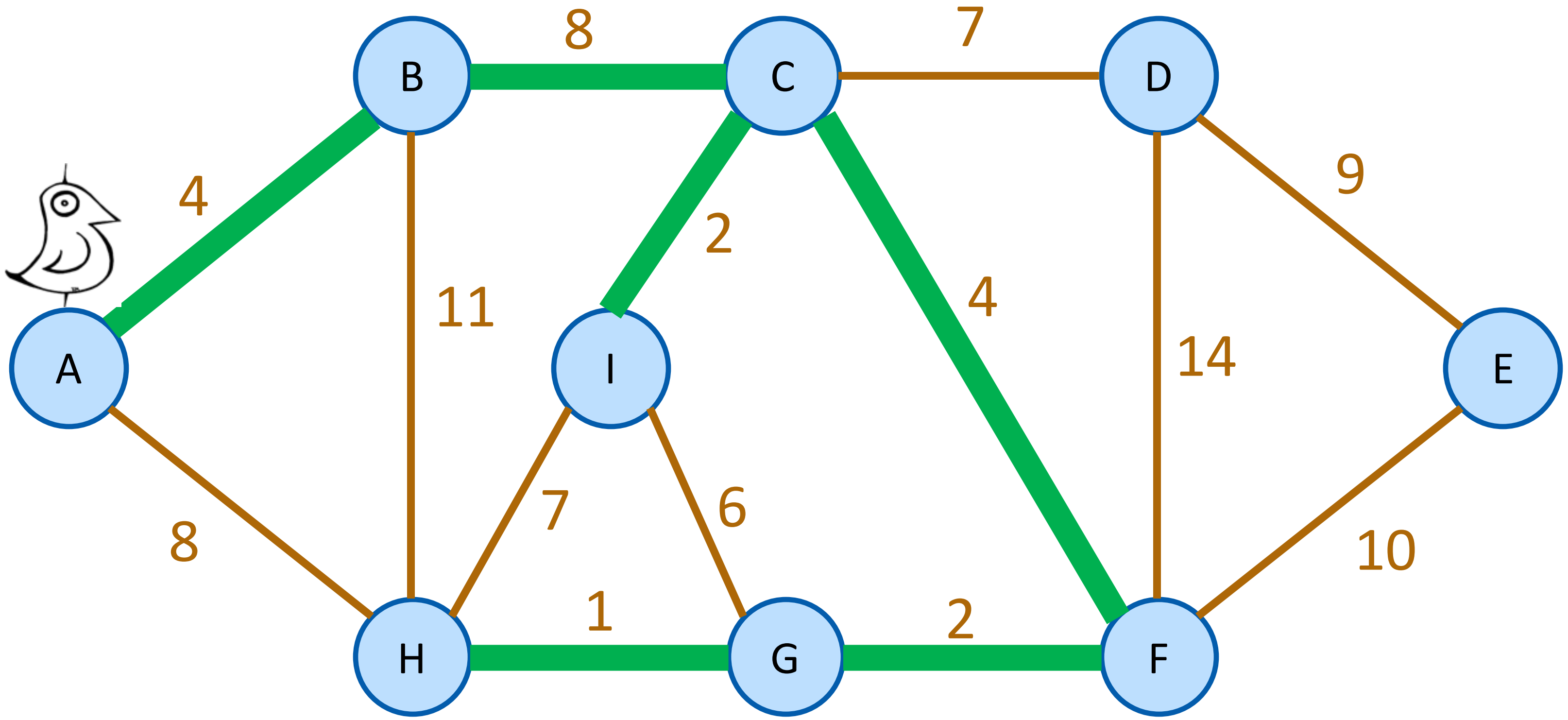
Start growing a tree, greedily add the shortest edge we can to grow the tree.



How to find an MST

Idea:

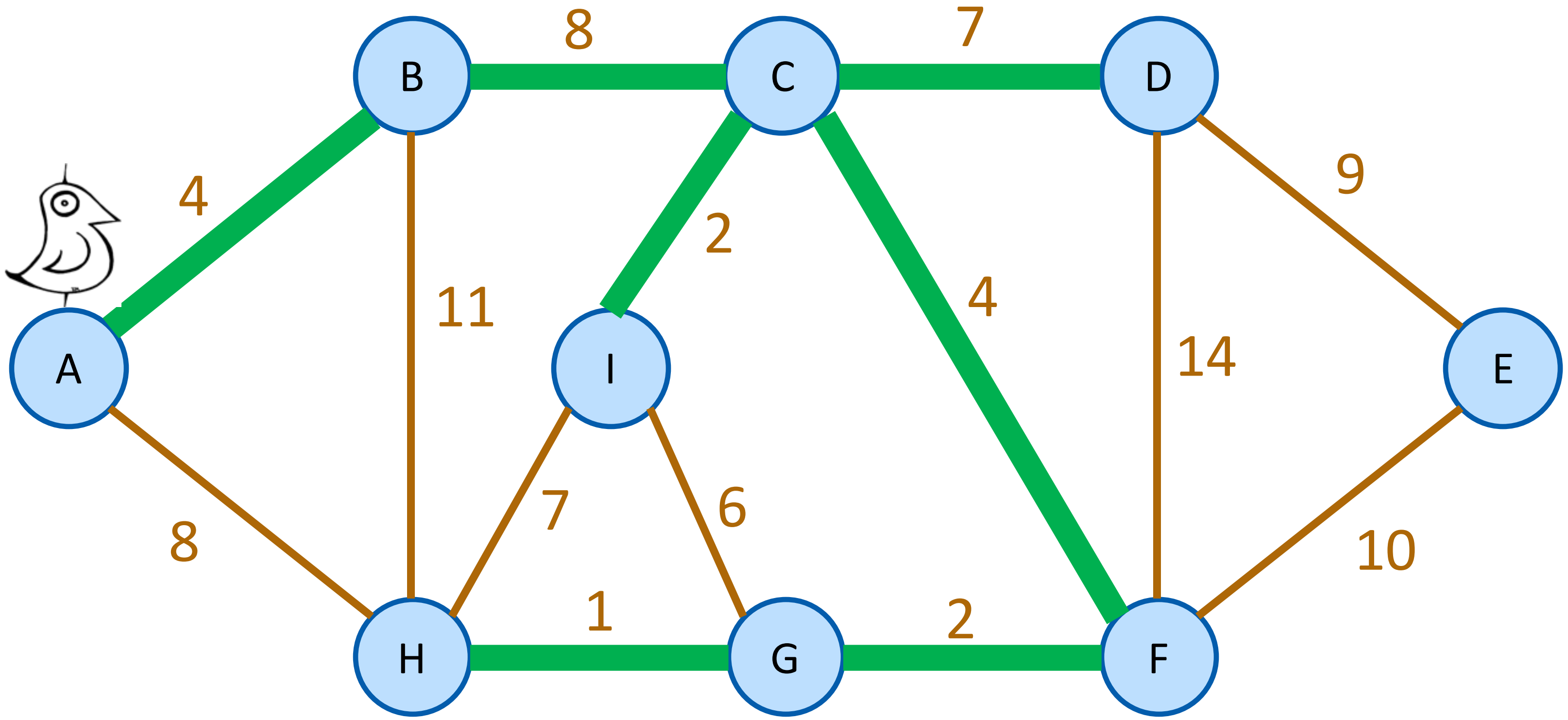
Start growing a tree, greedily add the shortest edge we can to grow the tree.



How to find an MST

Idea:

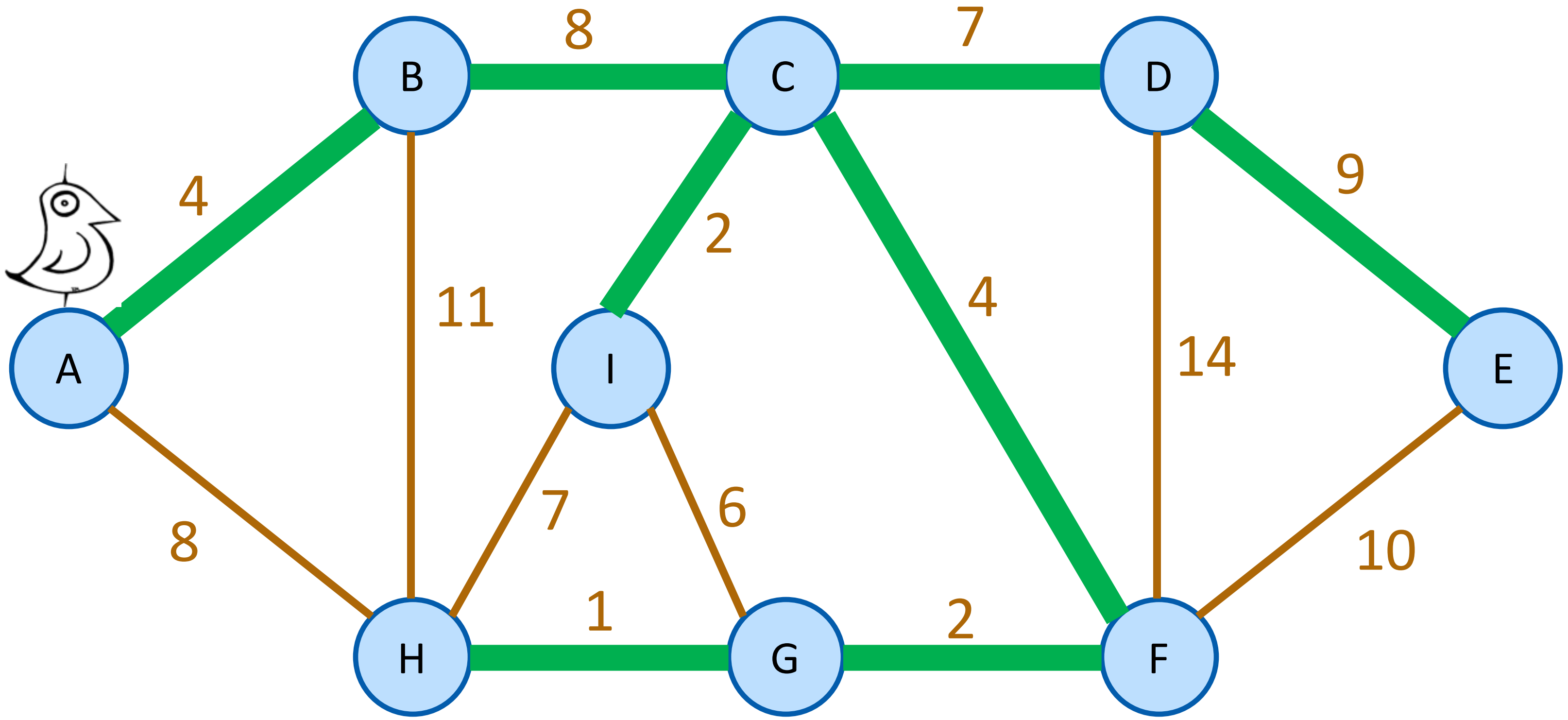
Start growing a tree, greedily add the shortest edge we can to grow the tree.



How to find an MST

Idea:

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



We’ve discovered Prim’s algorithm!

- slowPrim(G = (V,E), starting vertex s):
  - MST = {}
  - verticesVisited = { s }
  - **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

Naively, the running time is O(nm):

- For each of ≤n-1 iterations of the while loop:
  - Go through all the edges.

Prim's Algorithm

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"…

香港科技大学(广州)
THE HONG KONG
UNIVERSITY OF SCIENCE AND
TECHNOLOGY (GUANGZHOU)

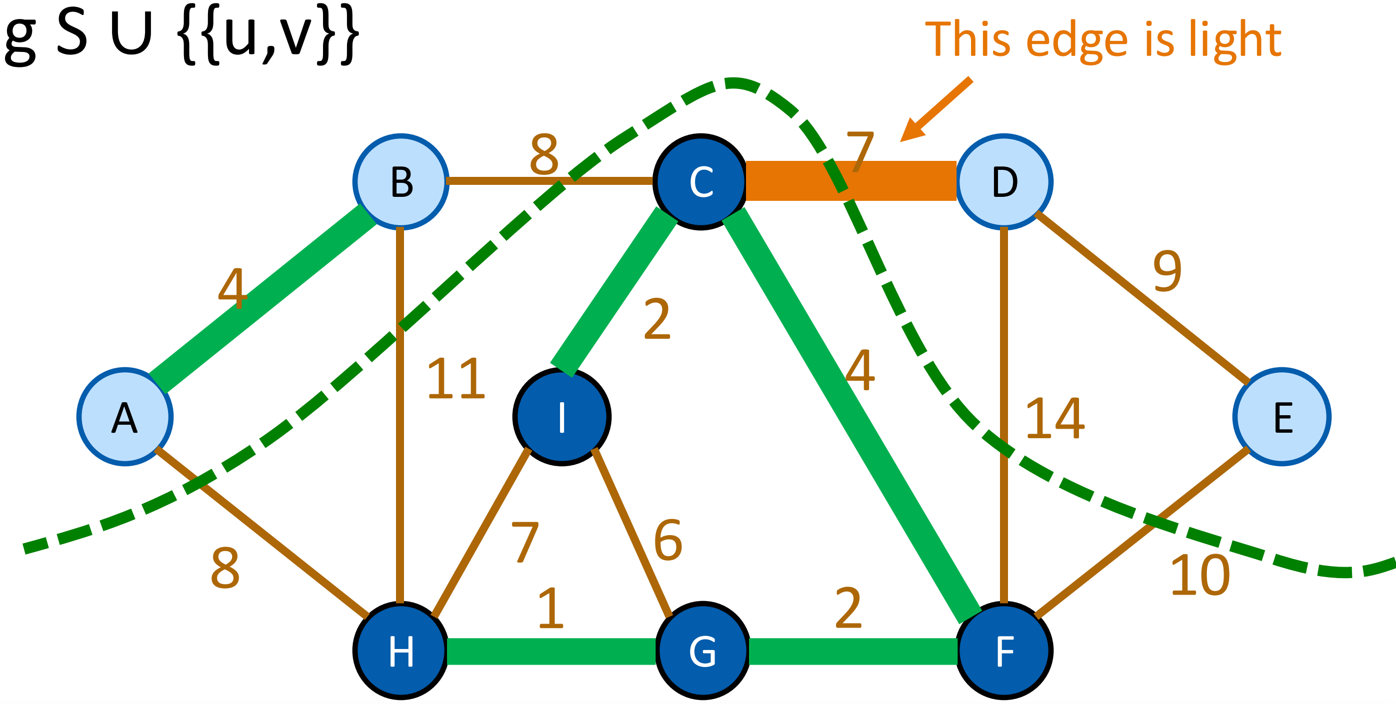
Does it work?

- 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}}

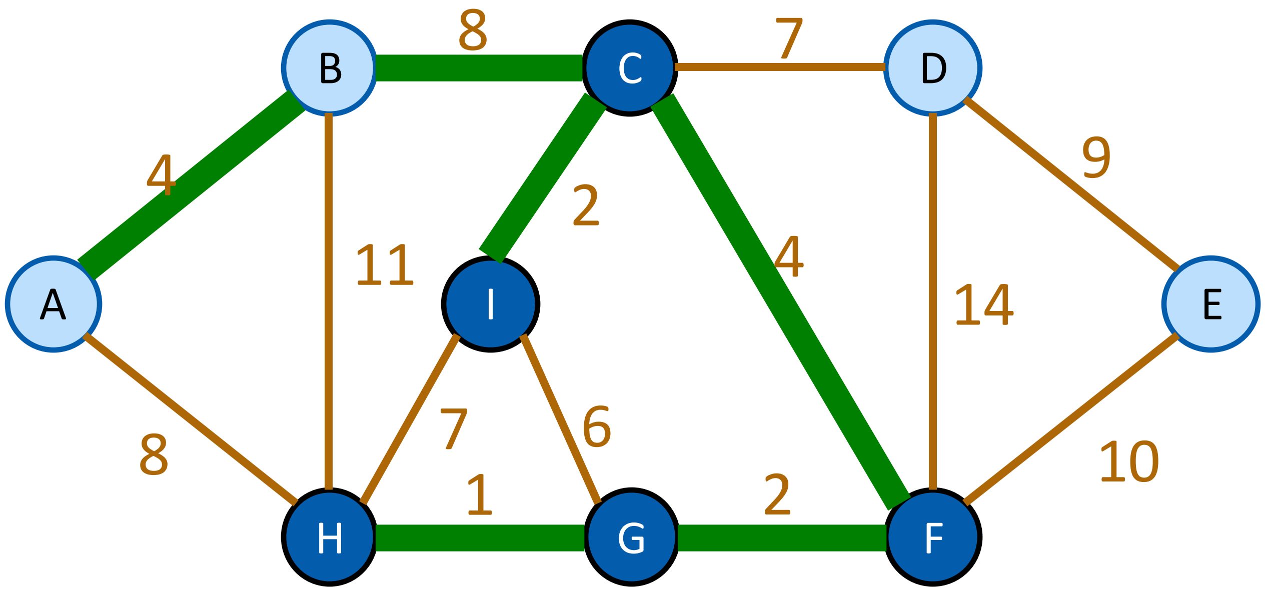


Prim's Algorithm

- 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?

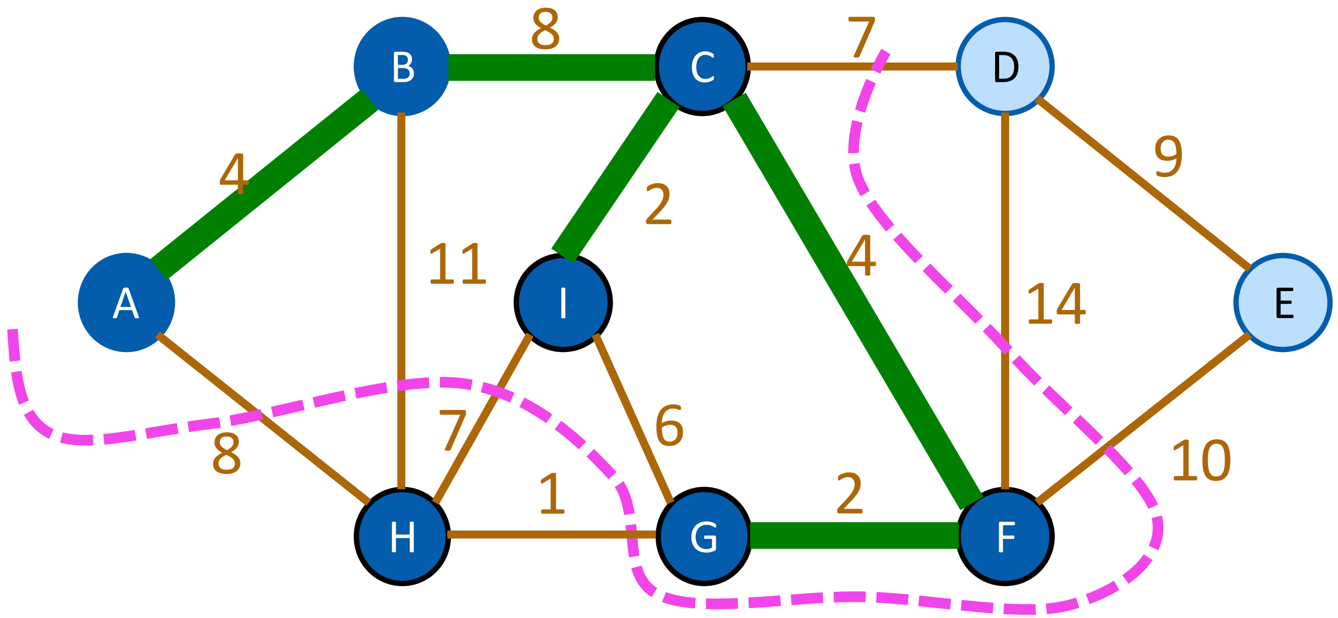
S is the set of edges selected so far



Prim's Algorithm

- 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.

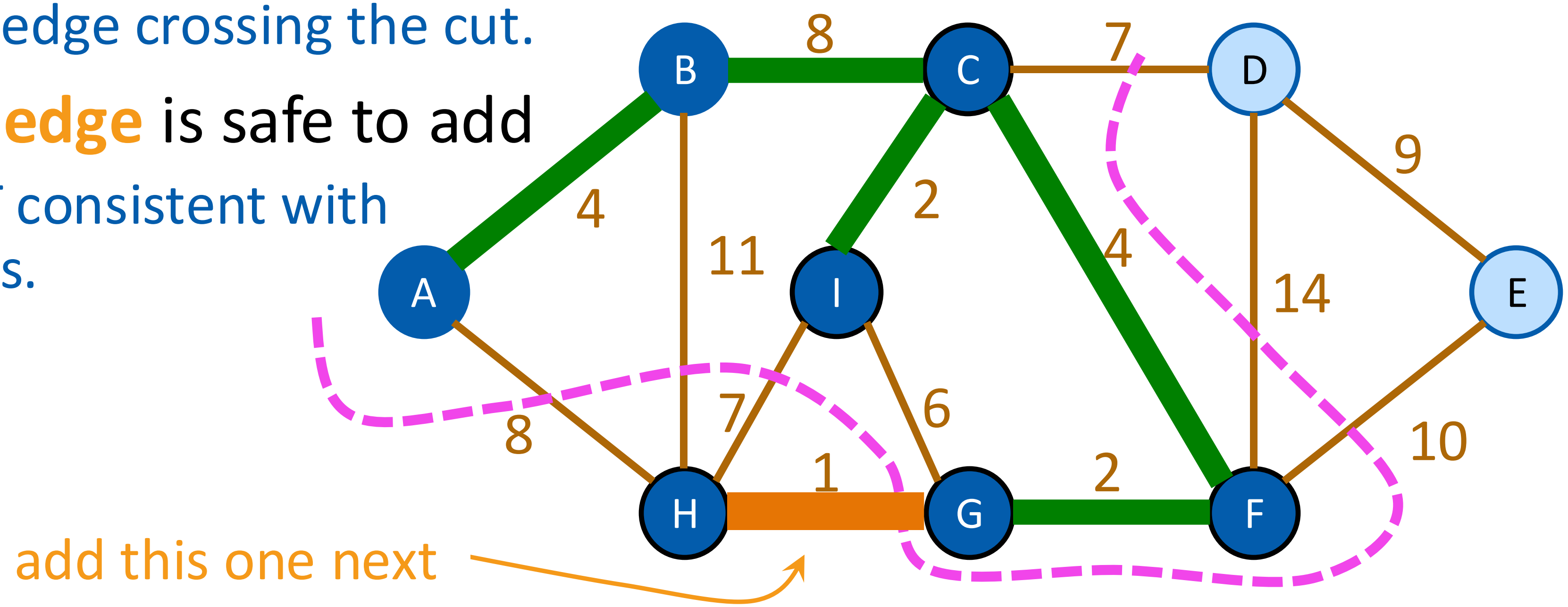
S is the set of edges selected so far



Prim's Algorithm

- 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



Prim's Algorithm

Formally,

- Inductive hypothesis:
  - After adding the t'th edge, there exists an MST with the edges added so far.
- Base case:
  - In the beginning, with no edges added, there exists an MST containing all the (zero) edges added so far. **YEP.**
- Inductive step:
  - If the inductive hypothesis holds for t (aka, the choices so far are safe), then it holds for t+1 (aka, the next edge we add is safe).
  - **That's what we just showed.**
- Conclusion:
  - After adding the n-1'st edge, there exists an MST with the edges added so far.
  - At this point, we have a spanning tree, so it better be a minimum spanning tree.

Prim's Algorithm

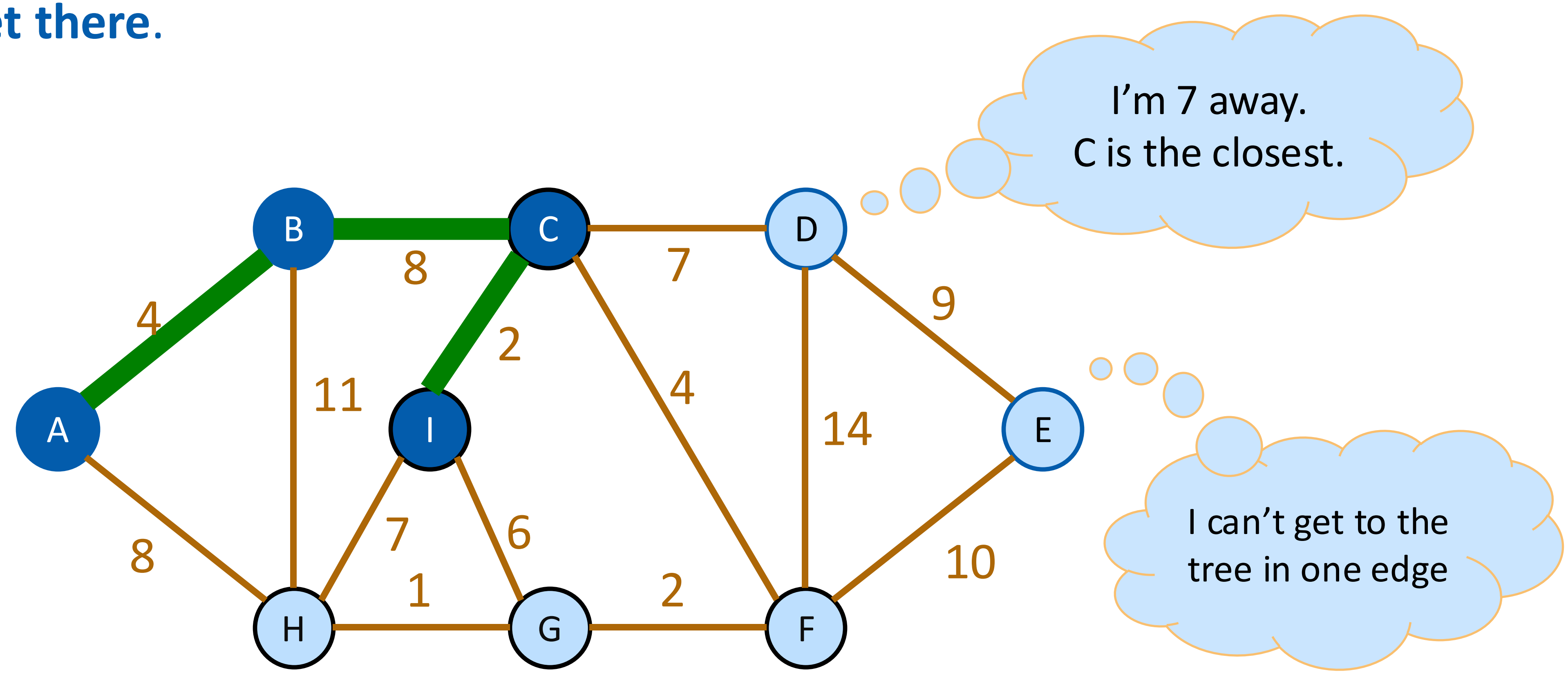
Two questions

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"…

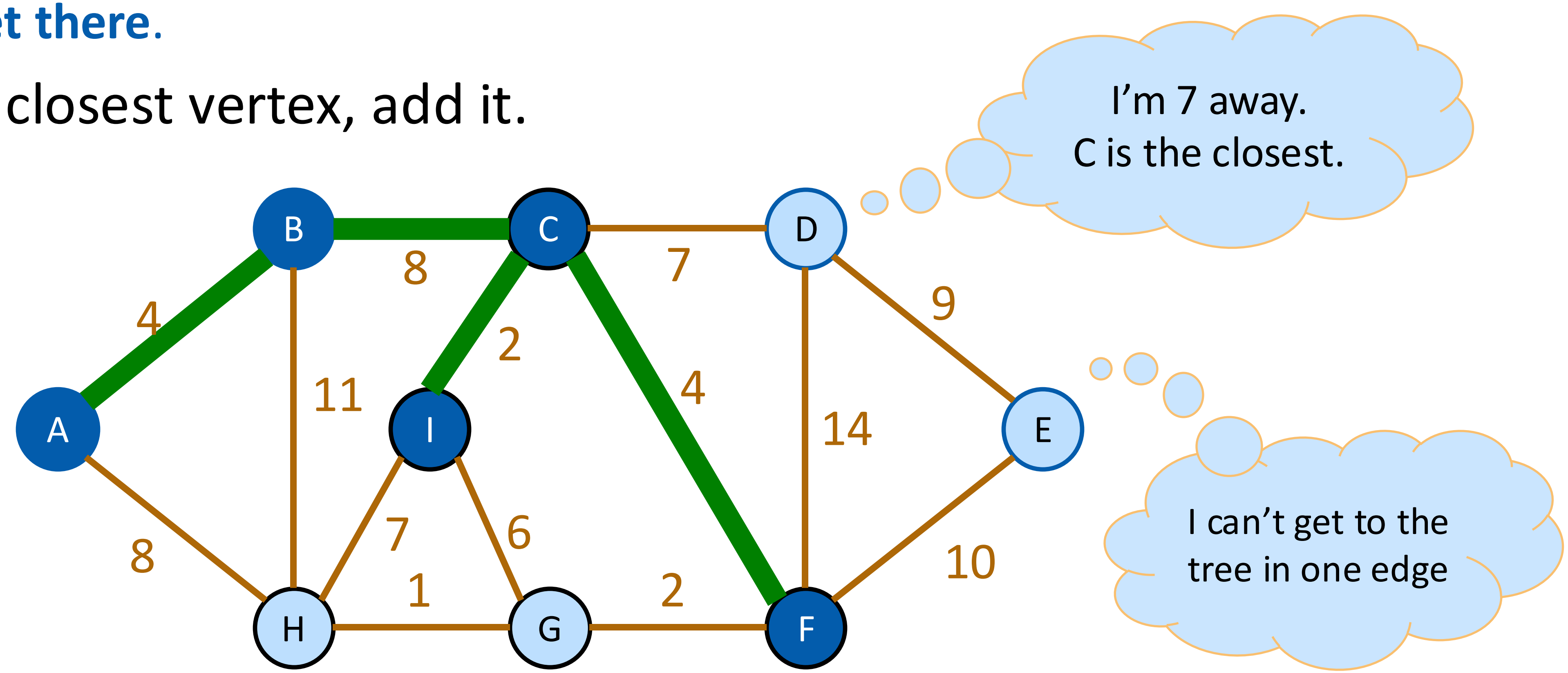
Efficient Implementation

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



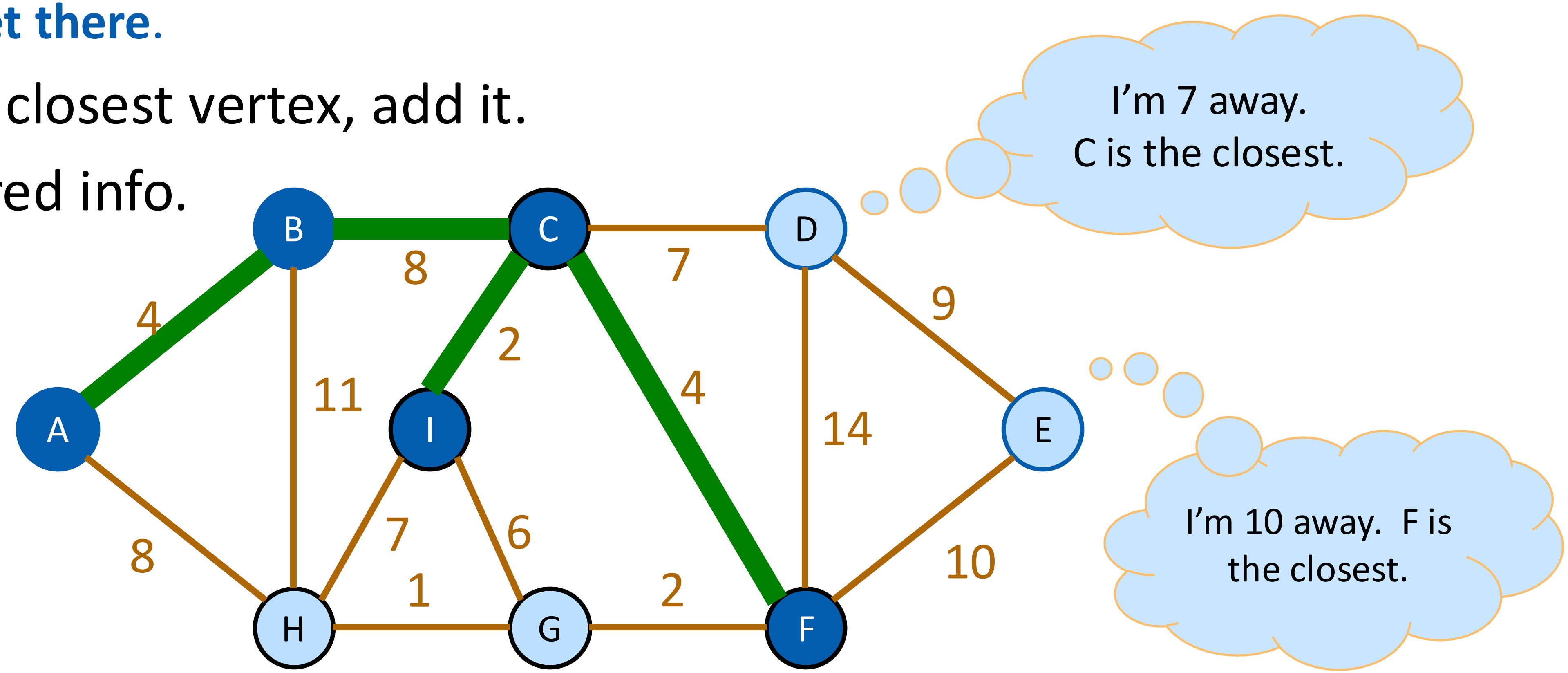
Efficient Implementation

- Each vertex keeps:
  - the **(single-edge) distance** from itself to the **growing spanning tree**
  - **how to get there**.
- Choose the closest vertex, add it.



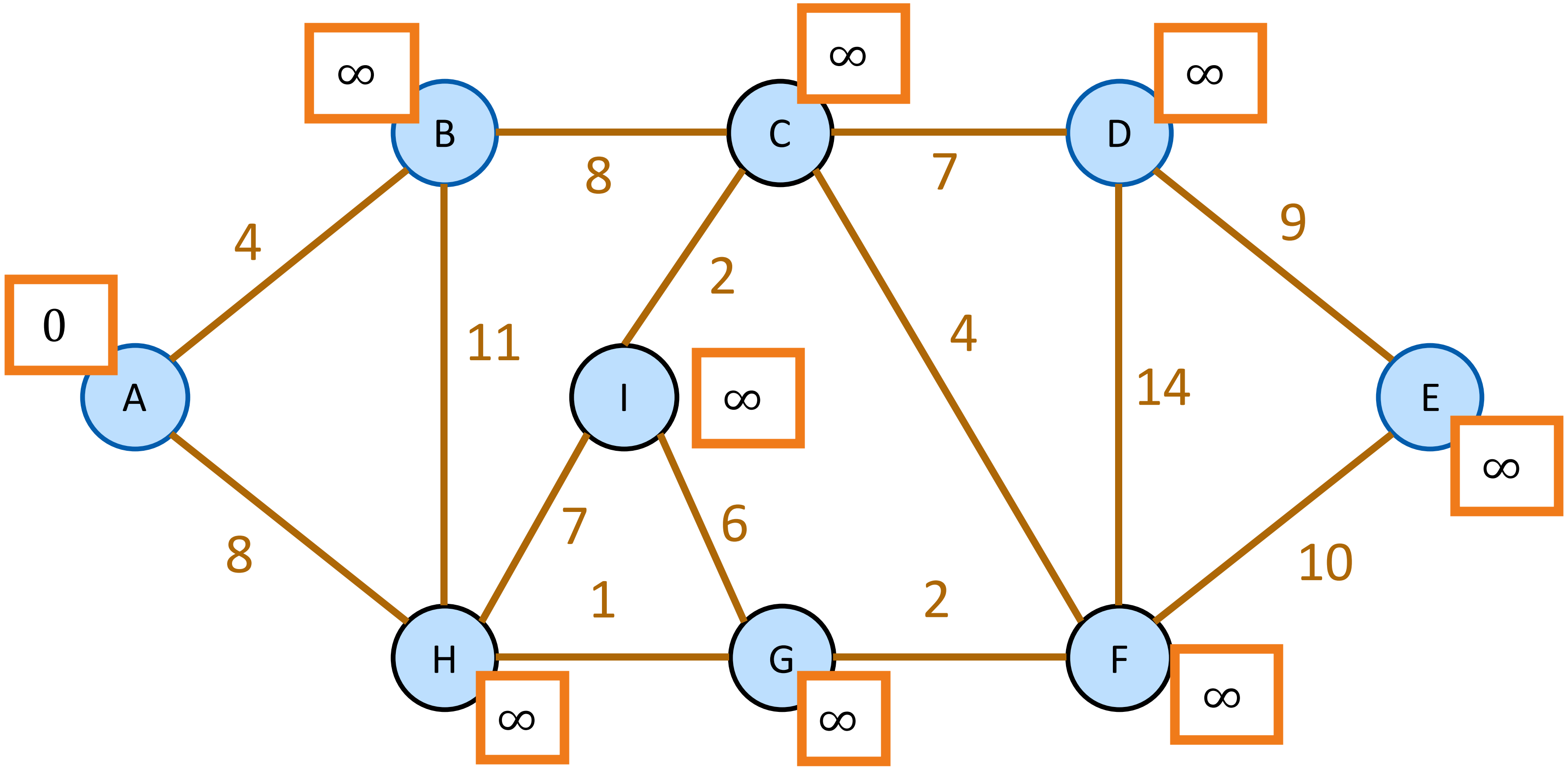
Efficient Implementation

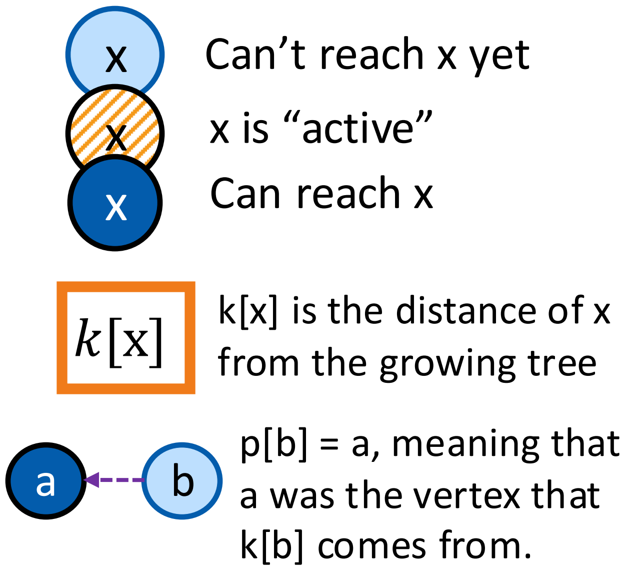
- Each vertex keeps:
  - the **(single-edge) distance** from itself to the **growing spanning tree**
  - **how to get there**.
- Choose the closest vertex, add it.
- Update stored info.



Efficient Implementation

Every vertex has a key and a parent



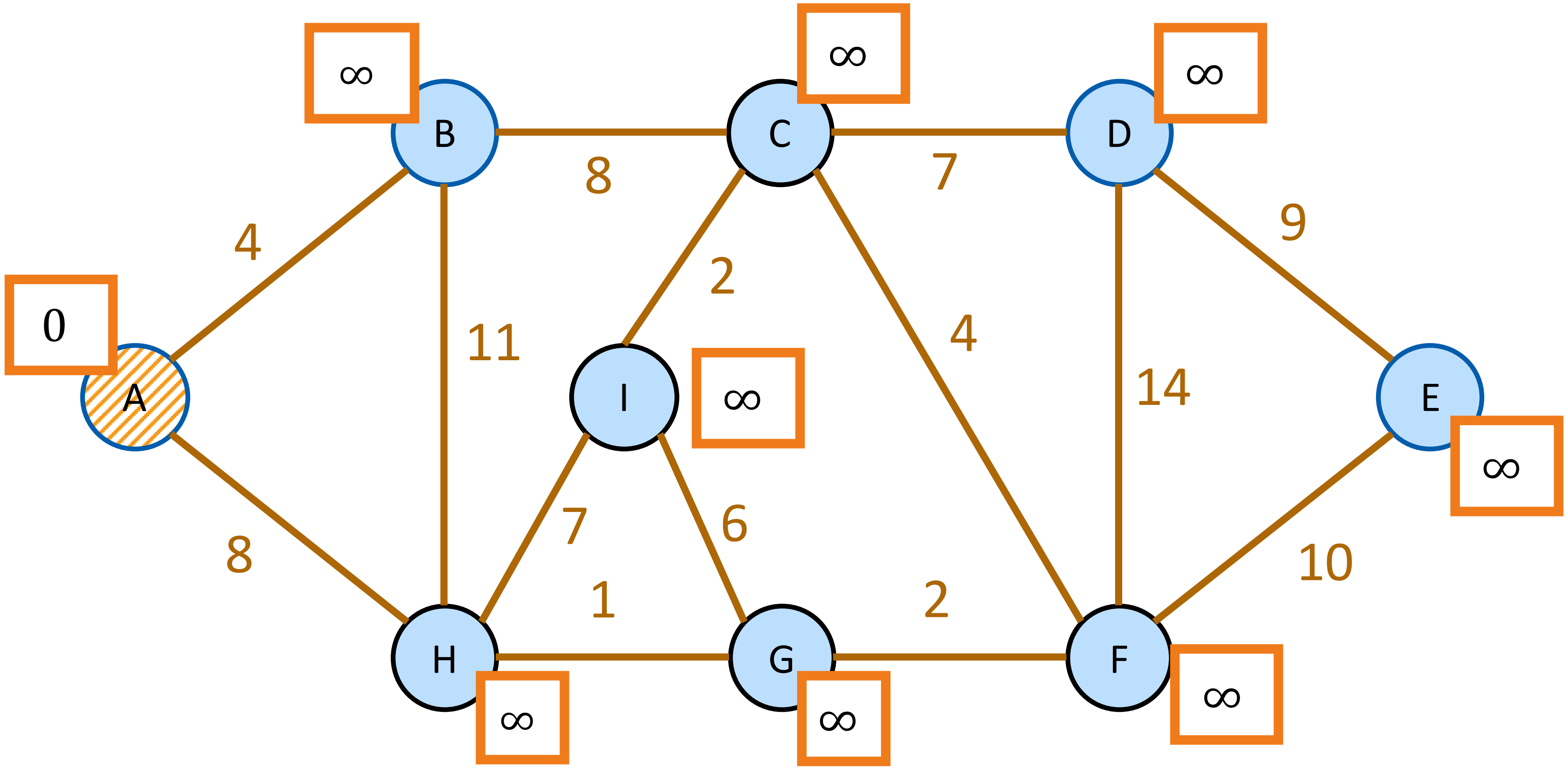


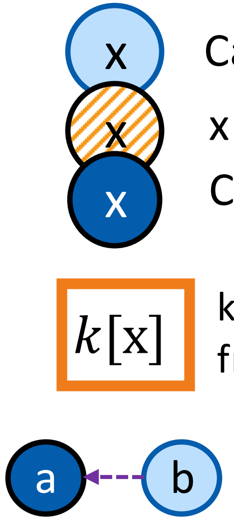
Until all the vertices are **reached:**

- 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

Efficient Implementation

Every vertex has a key and a parent





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.

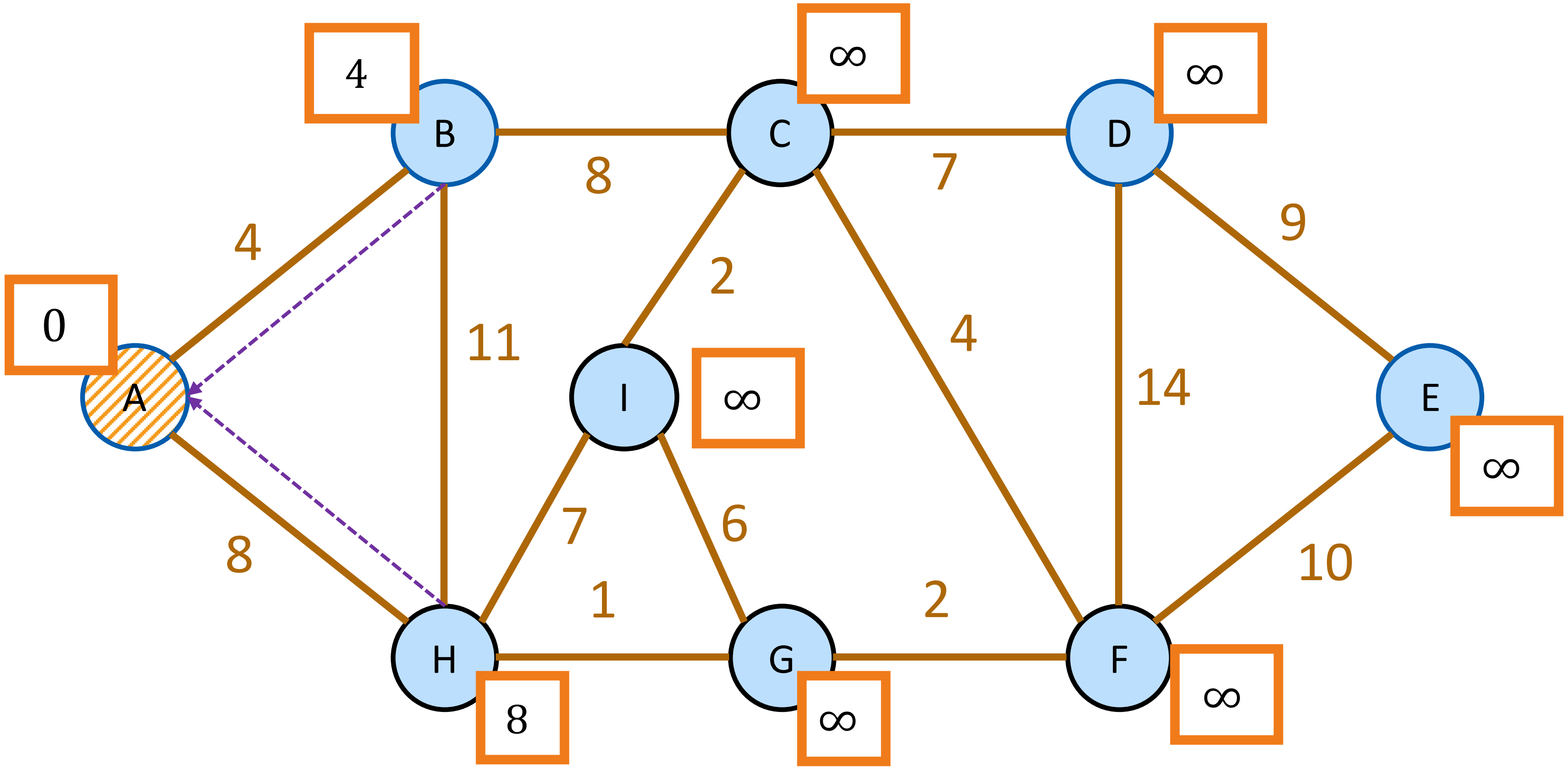
Until all the vertices are **reached:**

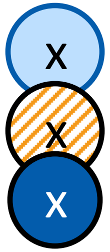
- Activate the **unreached** vertex u with the smallest key.
- **for each** of u's unreached neighbors v:
  - k[v] = min(k[v], weight(u,v))
  - if k[v] updated, p[v] = u

香港科技大学(广州) THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY (GUANGZHOU)

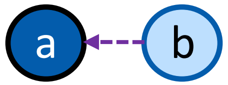
Efficient Implementation

Every vertex has a key and a parent





$k[\mathrm{x}]$ k[x] is the distance of x from the growing tree



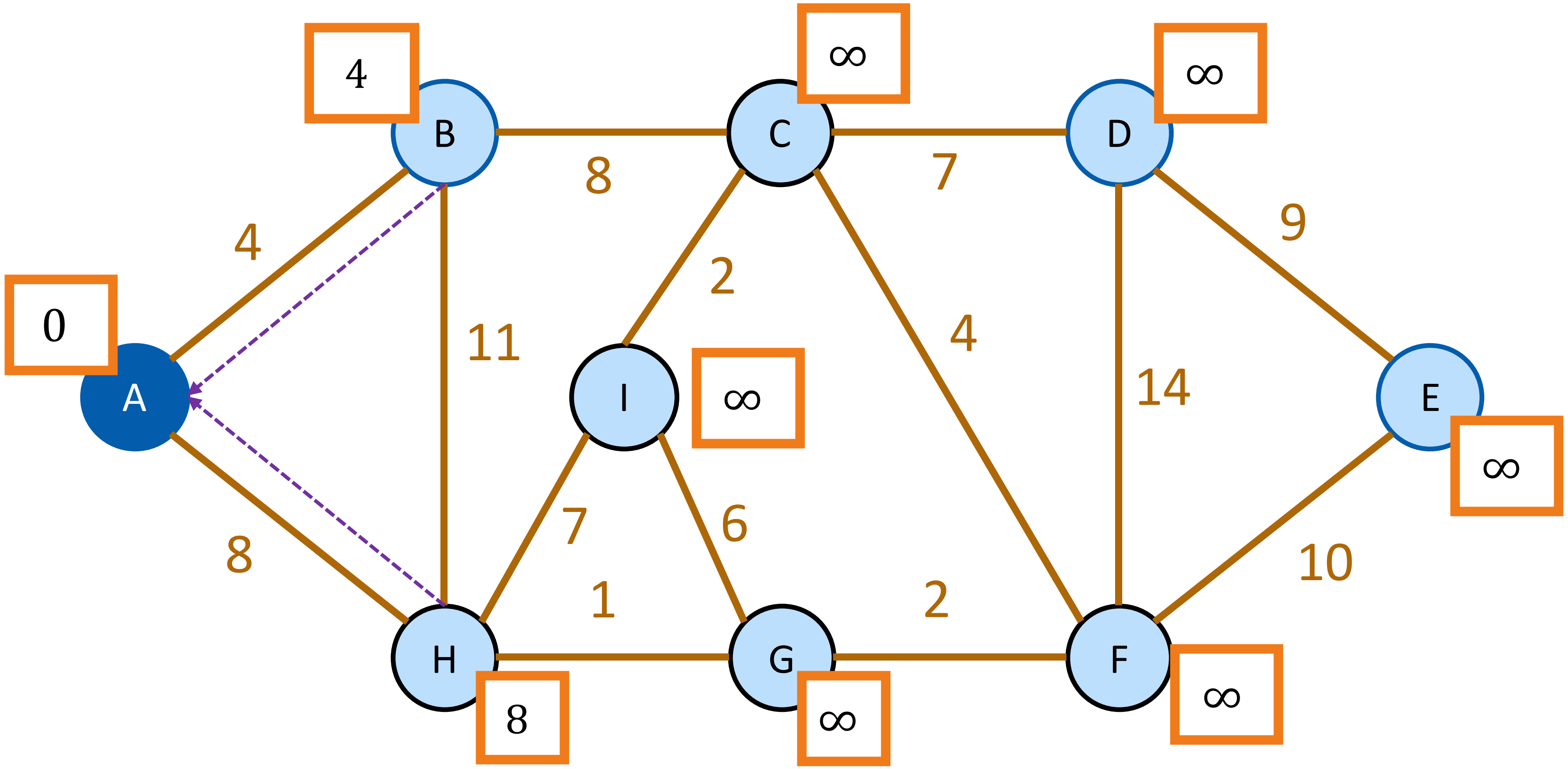
p[b] = a, meaning that a was the vertex that k[b] comes from.

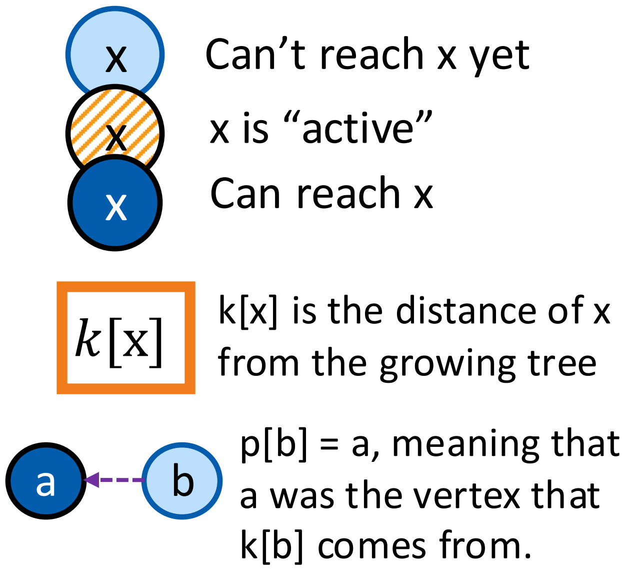
Until all the vertices are **reached:**

- 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

Efficient Implementation

Every vertex has a key and a parent





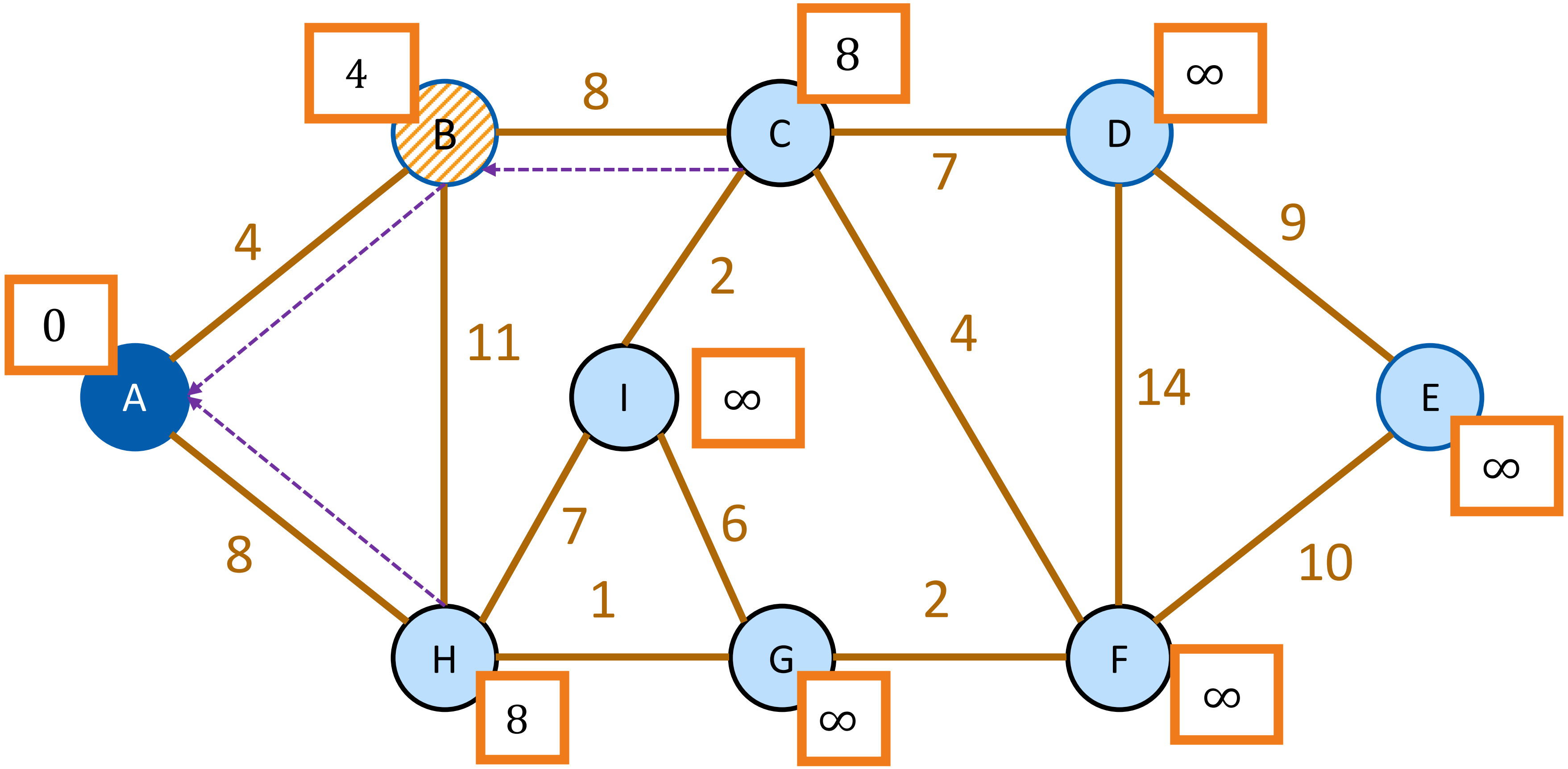
Until all the vertices are **reached:**

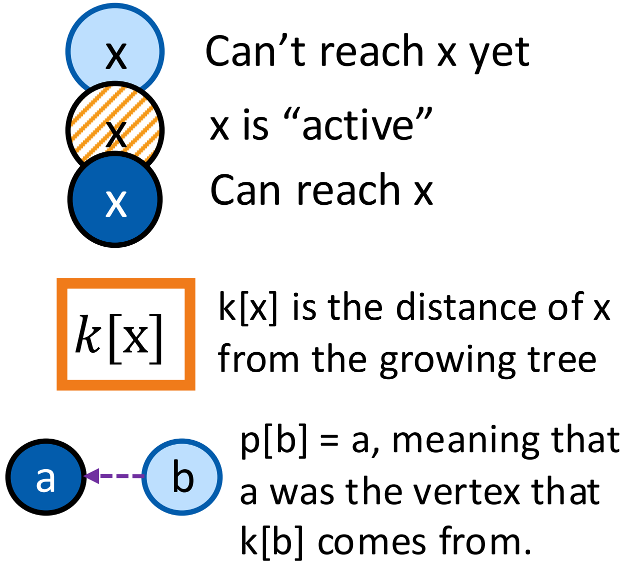
- Activate the **unreached** vertex u with the smallest key.
- **for each** of u's unreached neighbors v:
  - k[v] = min(k[v], weight(u,v))
  - if k[v] updated, p[v] = u

香港科技大学(广州) THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY (GUANGZHOU)

Efficient Implementation

Every vertex has a key and a parent





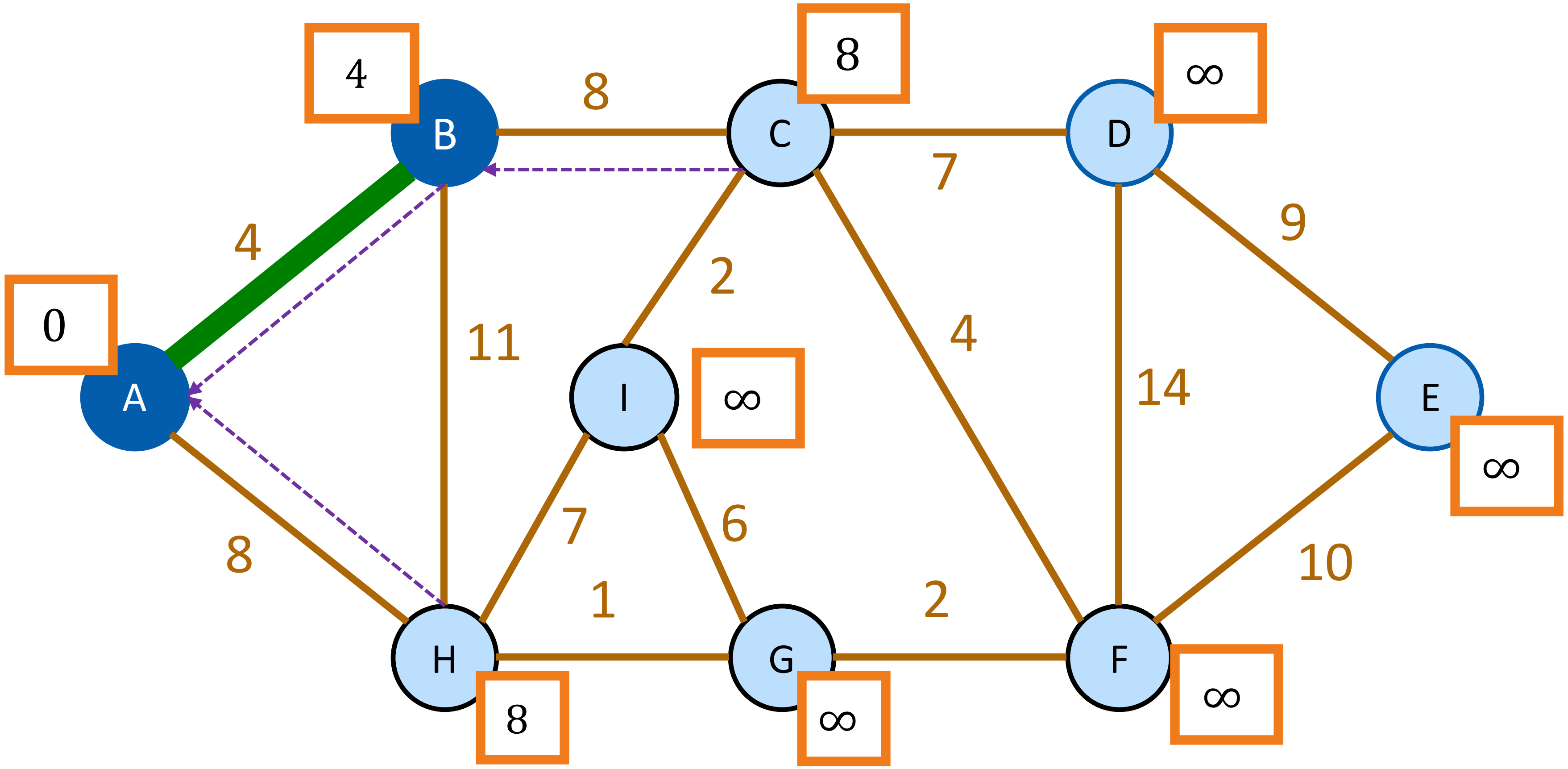
Until all the vertices are **reached:**

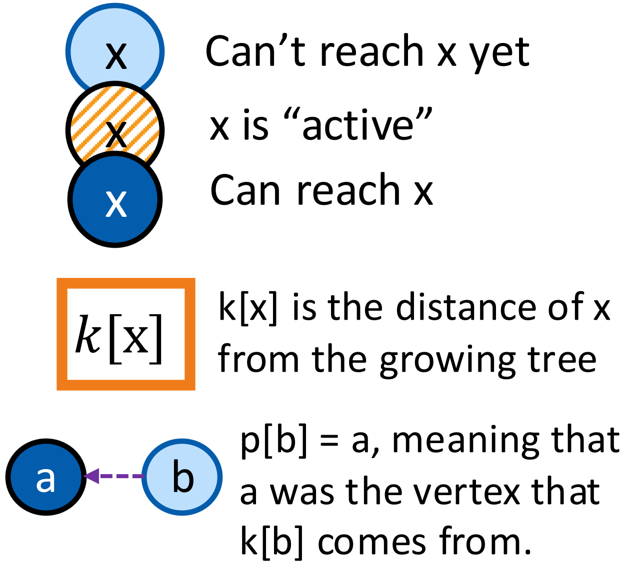
- Activate the unreached vertex u with the smallest key.
- **for each** of u's unreached neighbors v:
  - k[v] = min(k[v], weight(u,v))
  - if k[v] updated, p[v] = u

香港科技大学（广州）
THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY (GUANGZHOU)

Efficient Implementation

Every vertex has a key and a parent



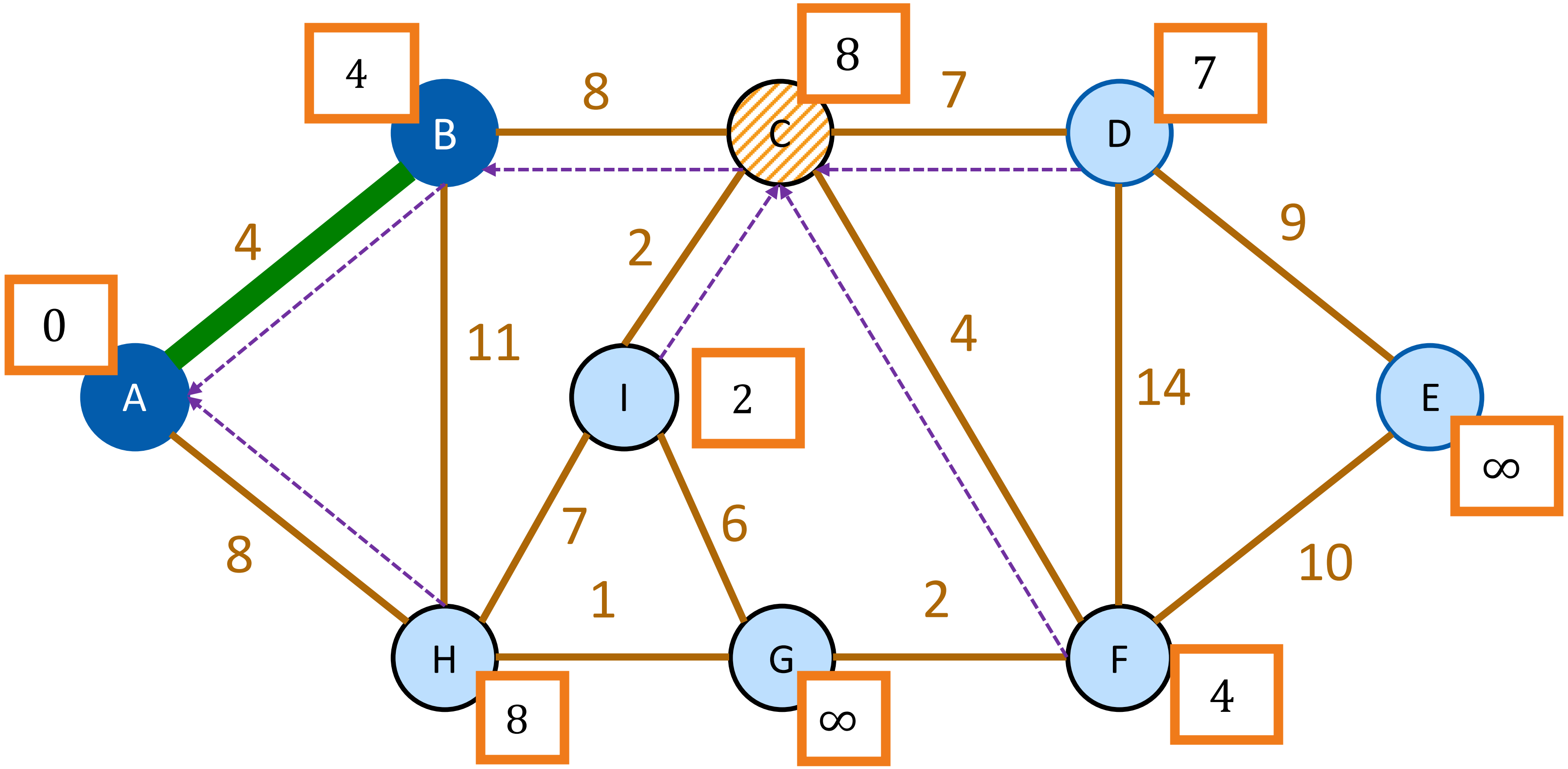


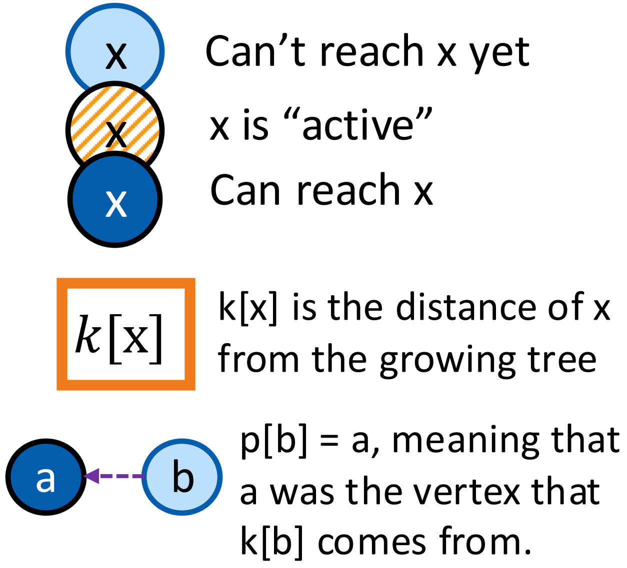
Until all the vertices are **reached:**

- 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

Efficient Implementation

Every vertex has a key and a parent





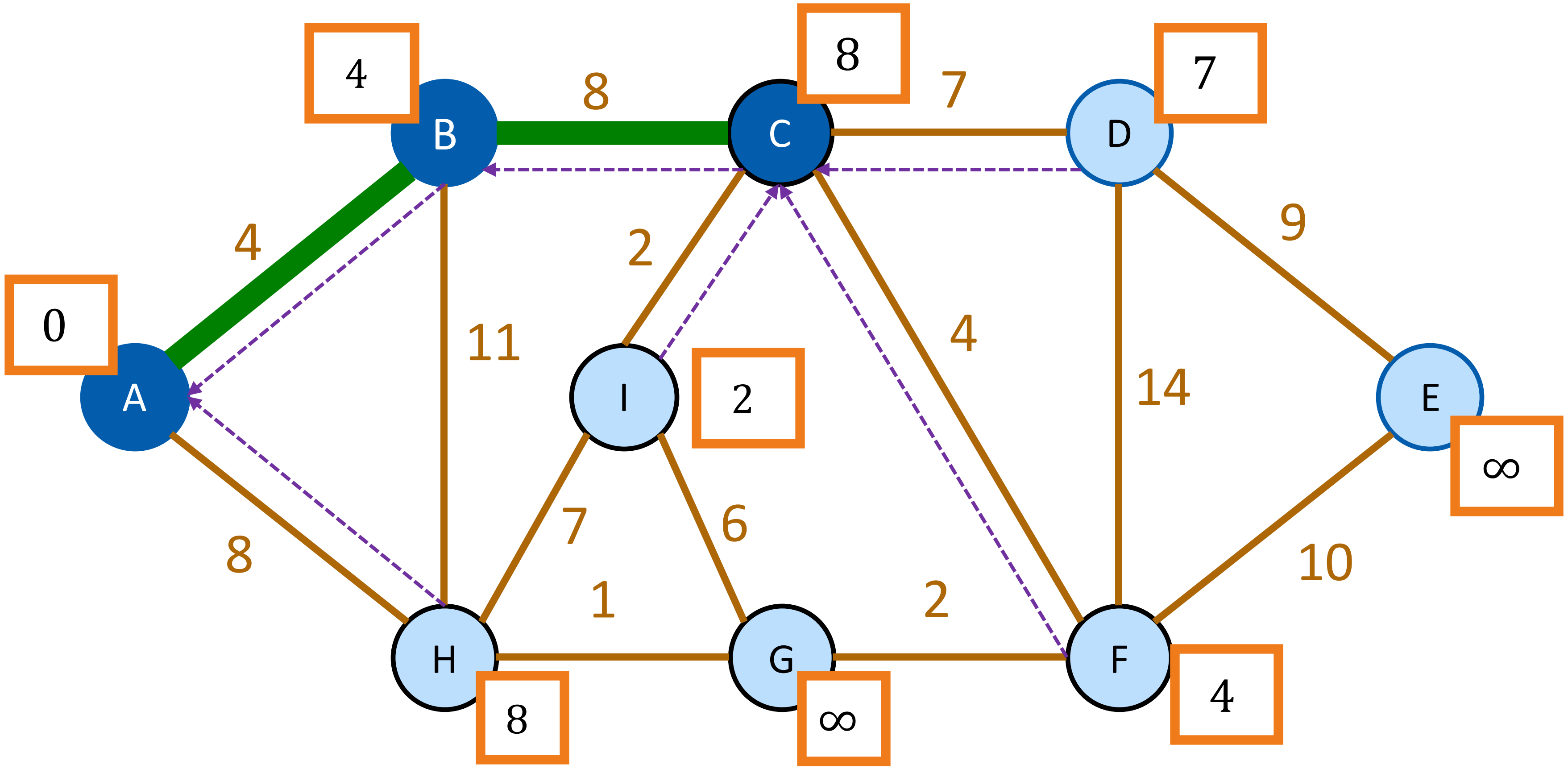
Until all the vertices are **reached:**

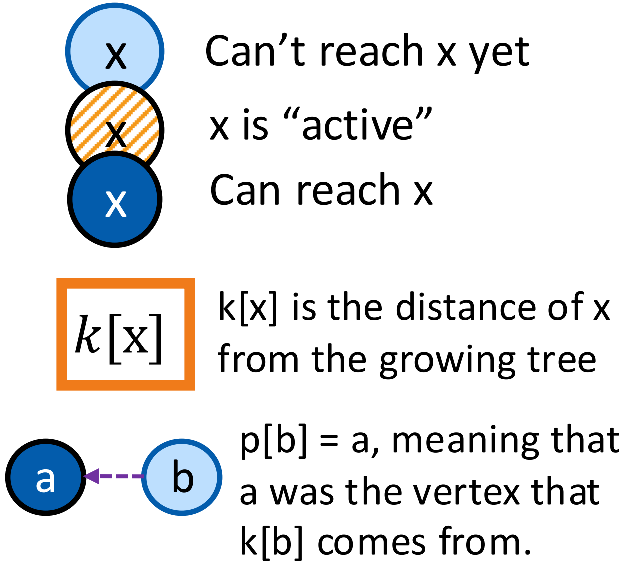
- Activate the **unreached** vertex u with the smallest key.
- **for each** of u's unreached neighbors v:
  - k[v] = min(k[v], weight(u,v))
  - if k[v] updated, p[v] = u

香港科技大学(广州)
THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY (GUANGZHOU)

Efficient Implementation

Every vertex has a key and a parent





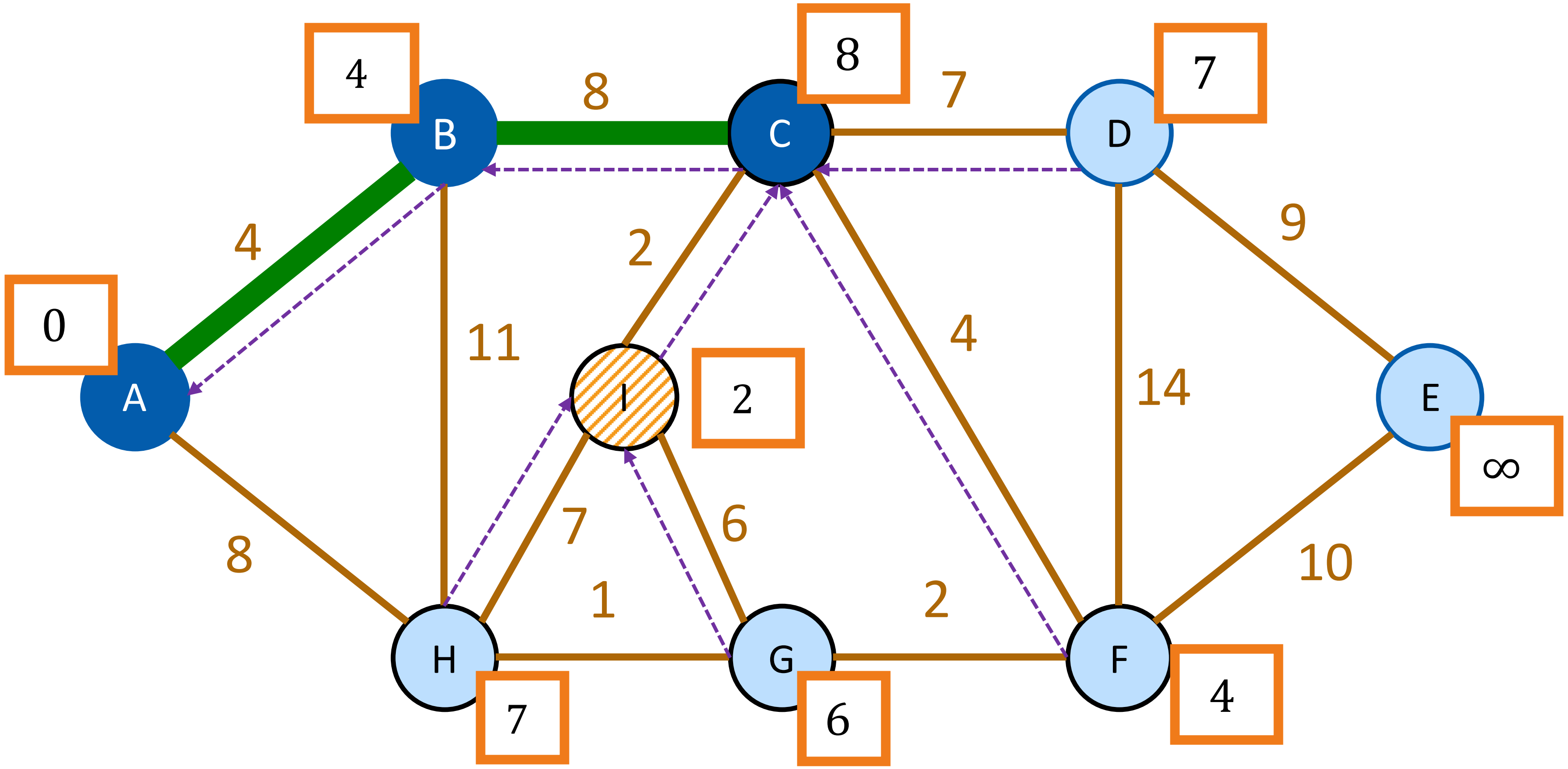
Until all the vertices are **reached:**

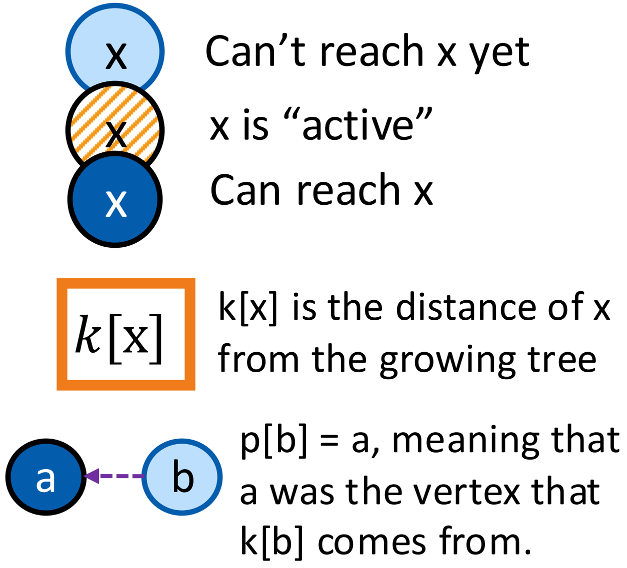
- Activate the unreached vertex u with the smallest key.
- **for each** of u's unreached neighbors v:
  - k[v] = min(k[v], weight(u,v))
  - if k[v] updated, p[v] = u

香港科技大学（广州）
THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY (GUANGZHOU)

Efficient Implementation

Every vertex has a key and a parent



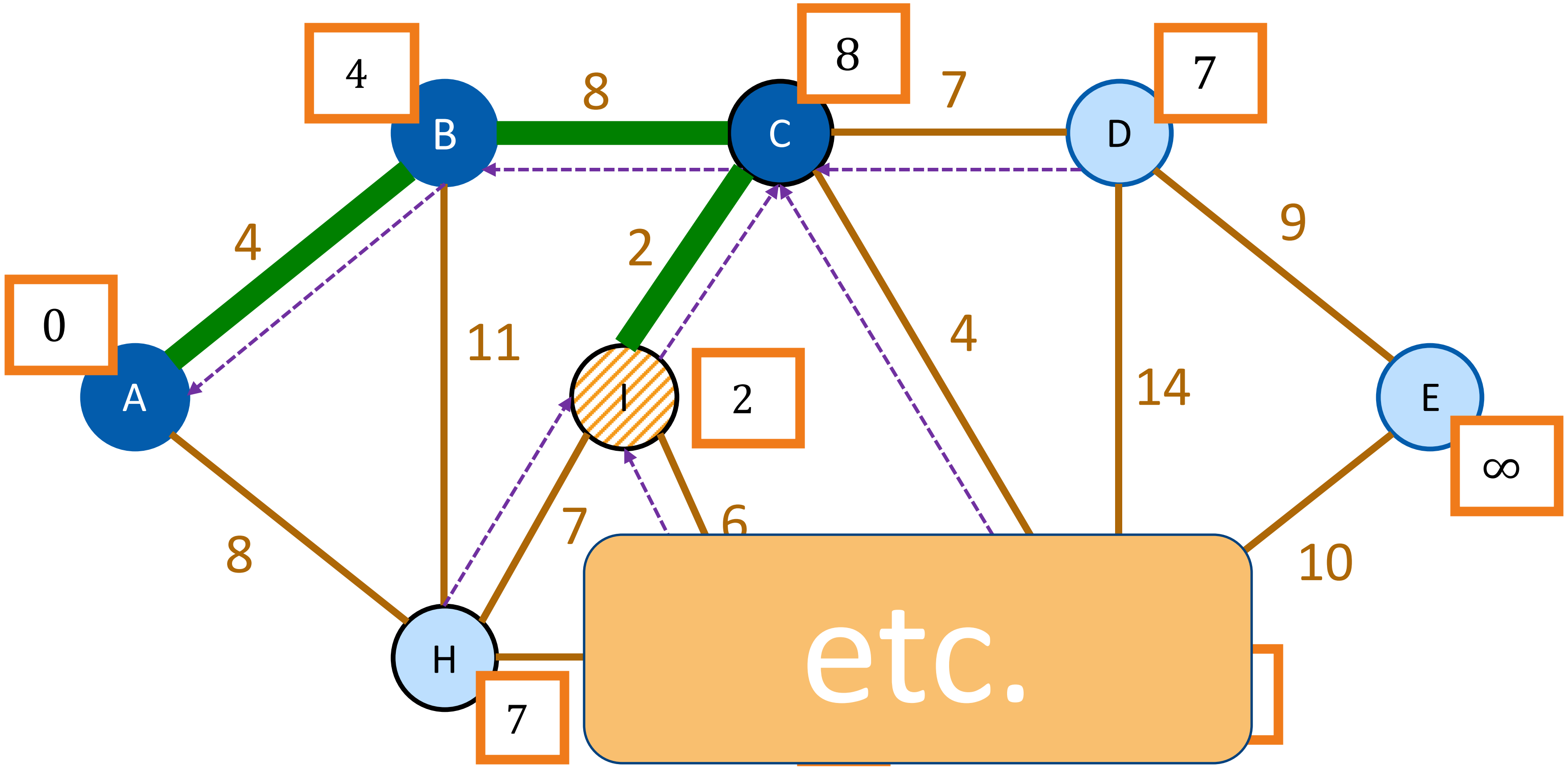


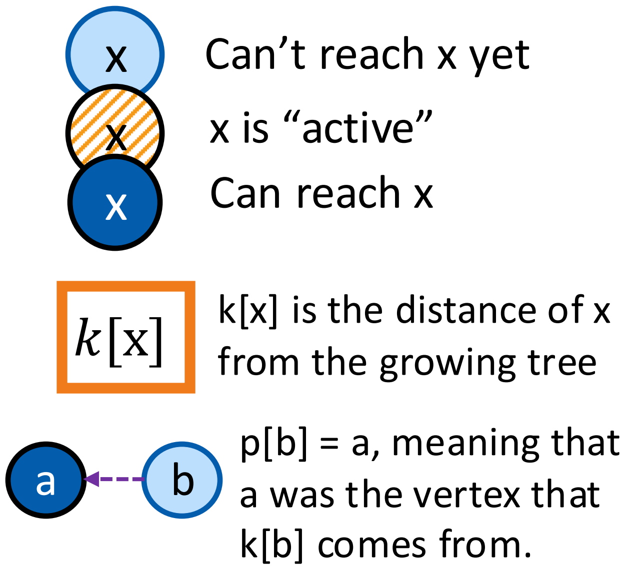
Until all the vertices are **reached:**

- 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

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

Prim's Algorithm

- Very similar to Dijkstra's algorithm!
- **Differences:**
  1. Keep track of p[v] in order to return a tree at the end
     - But Dijkstra's can do that too, that's not a big difference.
  2. Instead of d[v] which we update by
     - d[v] = min(d[v], d[u] + w(u,v))

     we keep k[v] which we update by
     - k[v] = min(k[v], w(u,v))

Thing 2 is the main difference.

Prim's Algorithm

Two questions

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

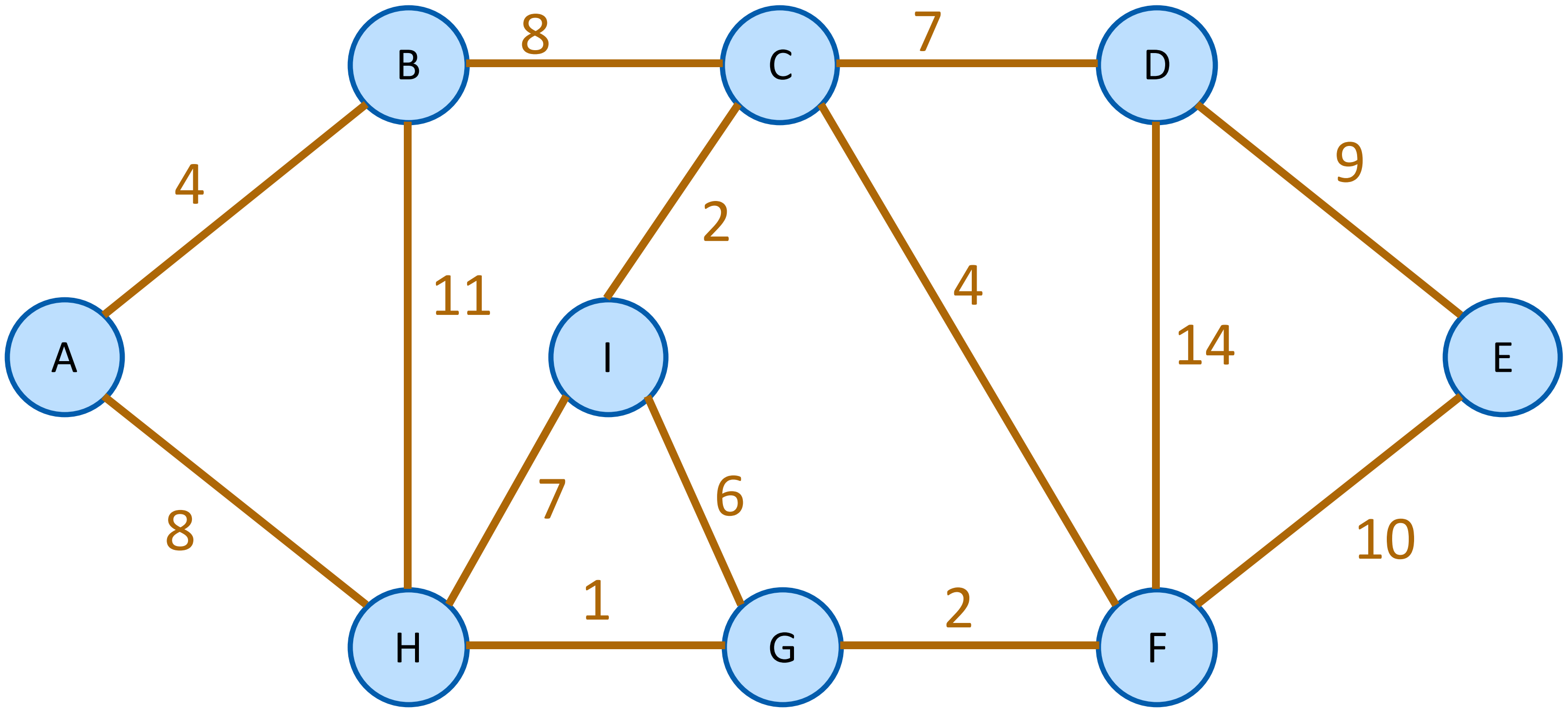
2. How do we actually implement this?
   - the pseudocode above says "slowPrim"…
     - **Implement it basically the same way we'd implement Dijkstra!**

Prim's Algorithm

That's not the only greedy algorithm for MST!

Thoughts

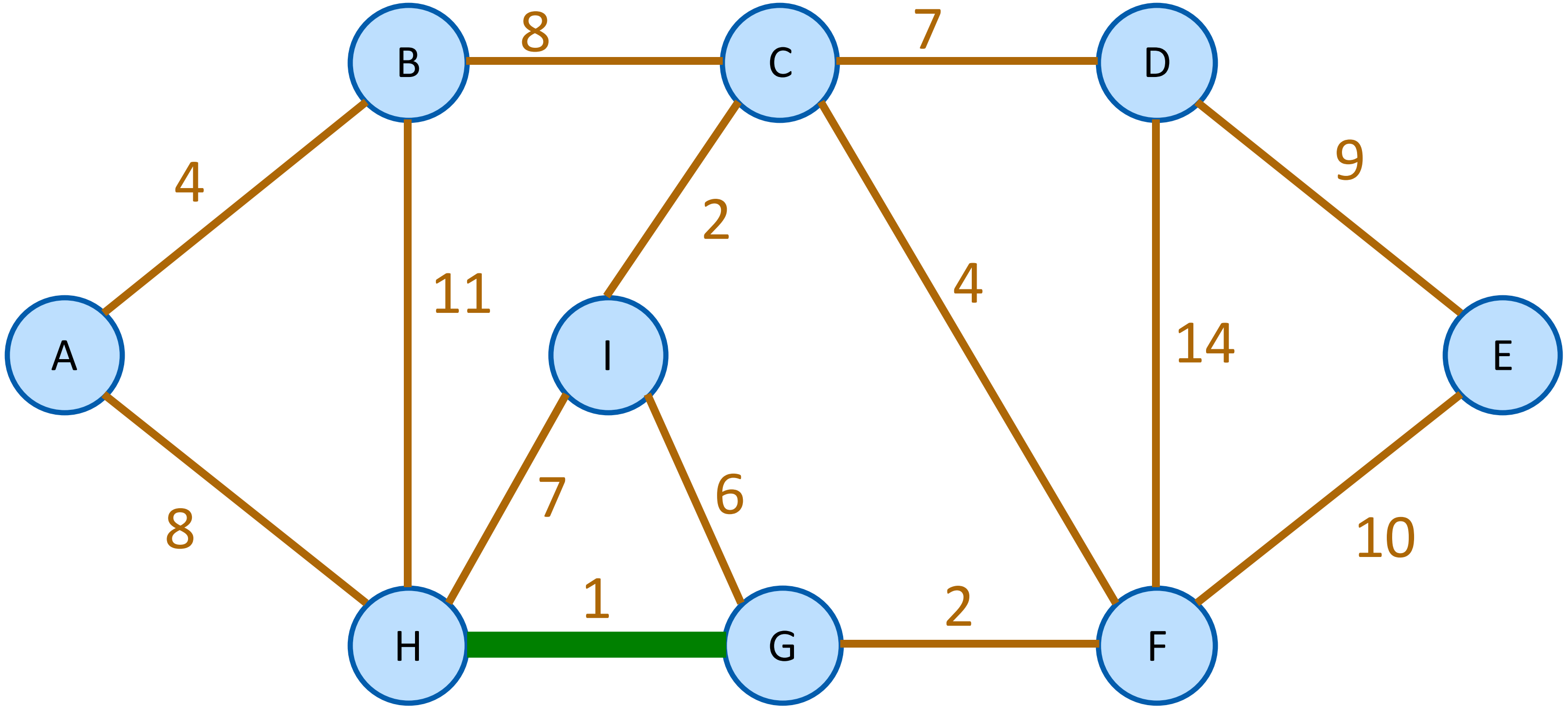
what if we just always take the cheapest edge?
whether or not it's connected to what we have so far?



Thoughts

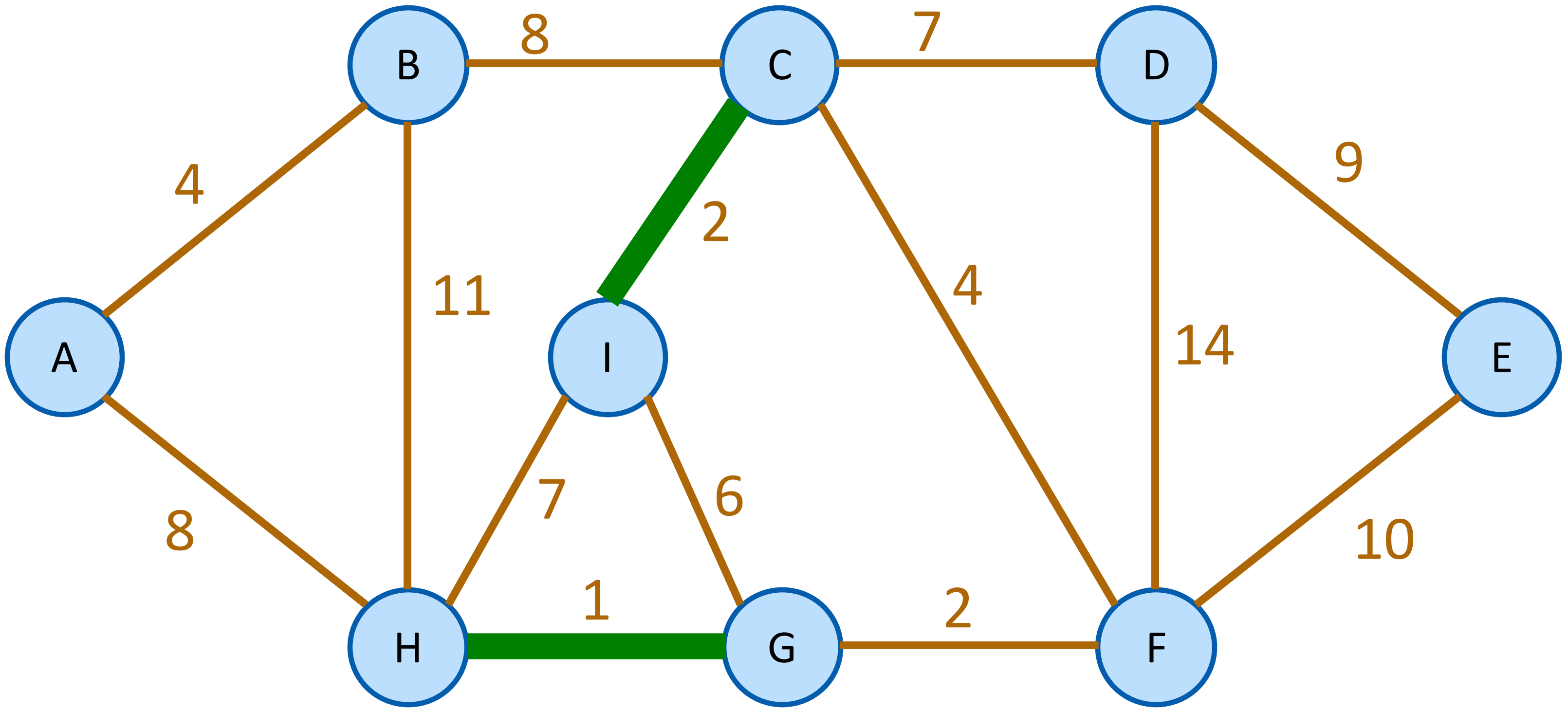
what if we just always take the cheapest edge?

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



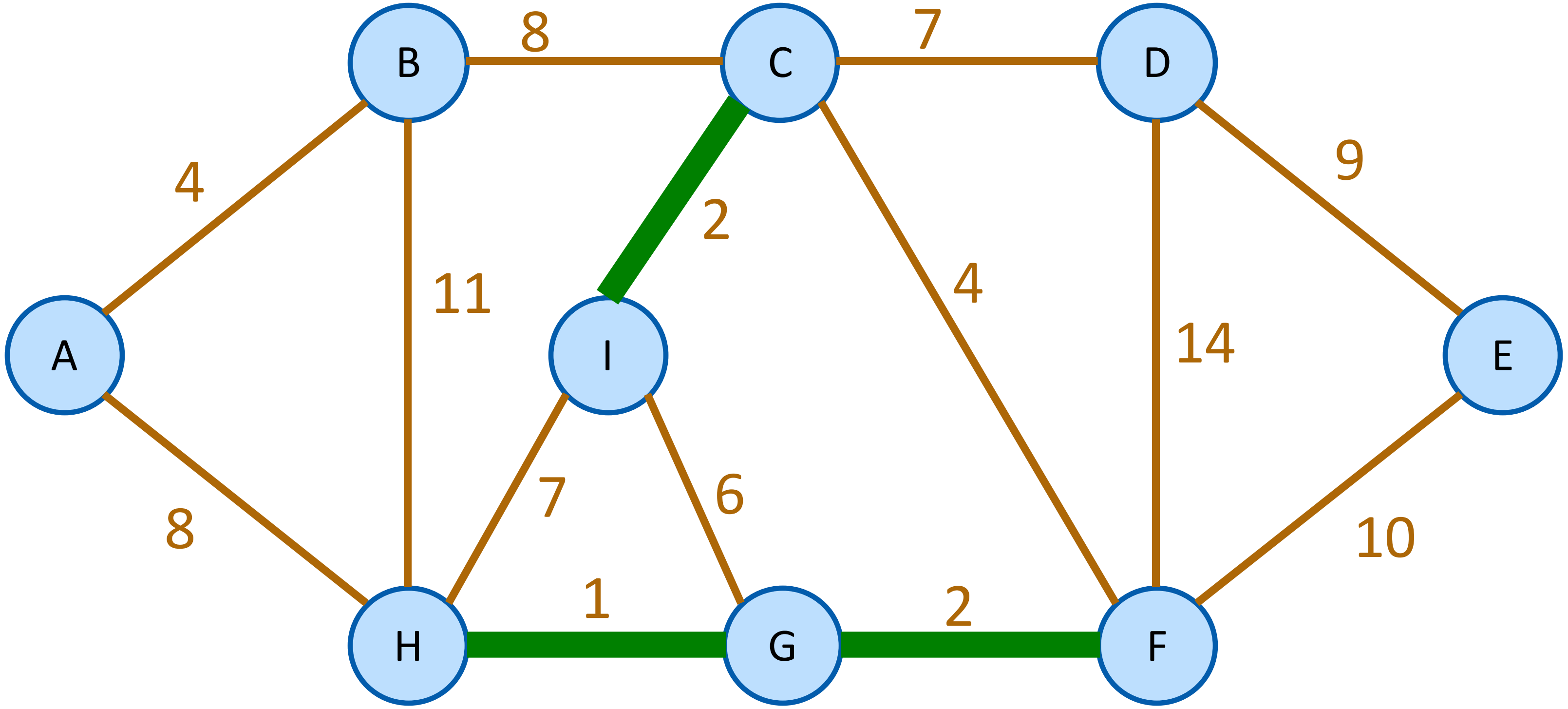
Thoughts

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



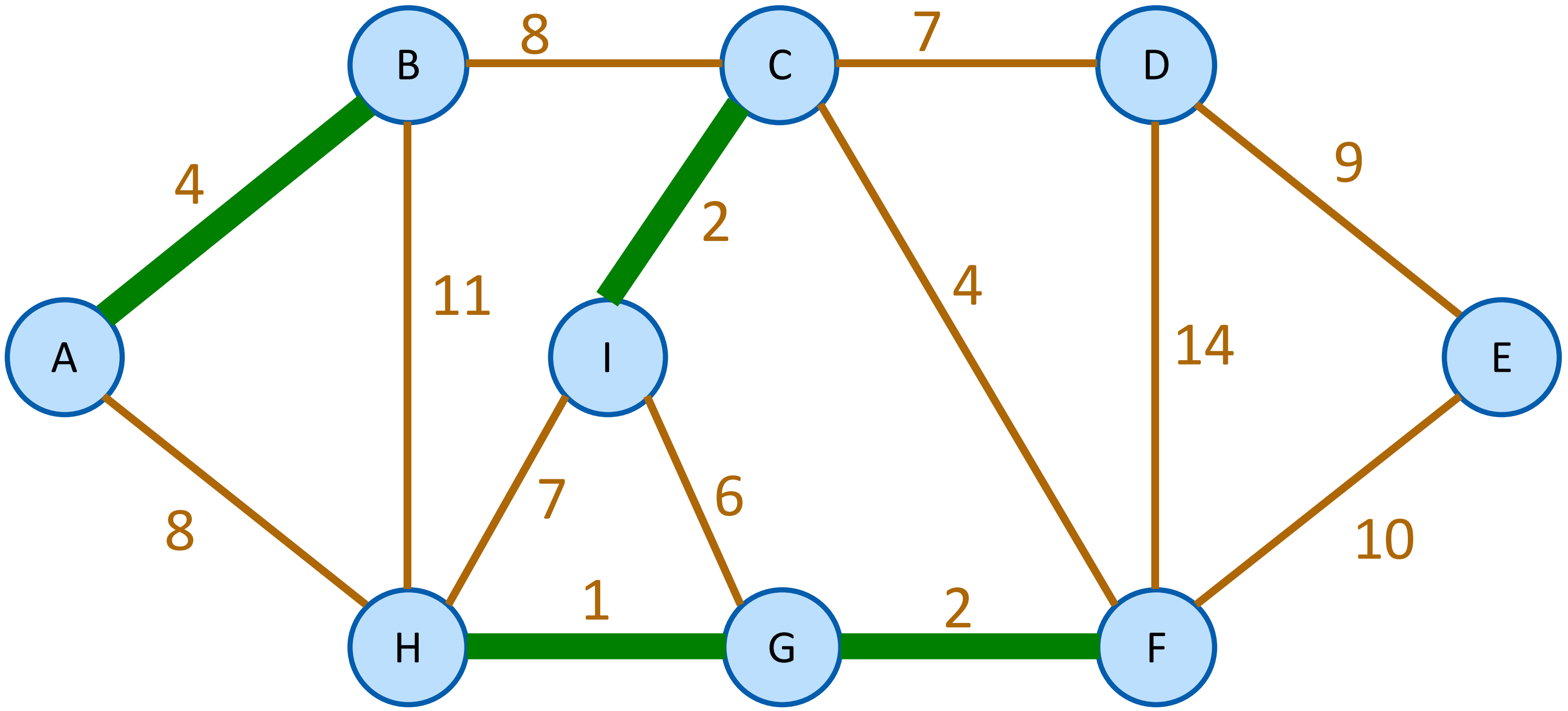
Thoughts

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



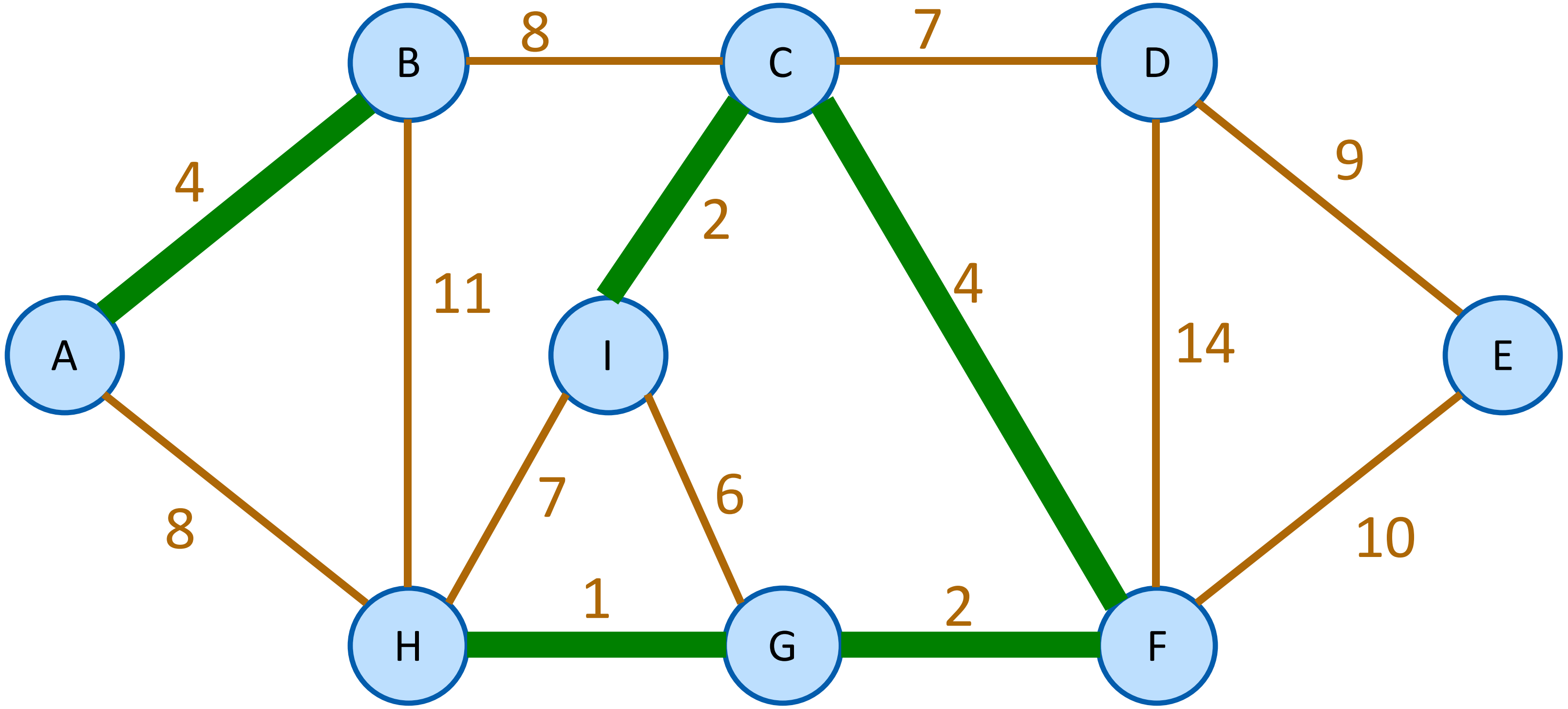
Thoughts

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



Thoughts

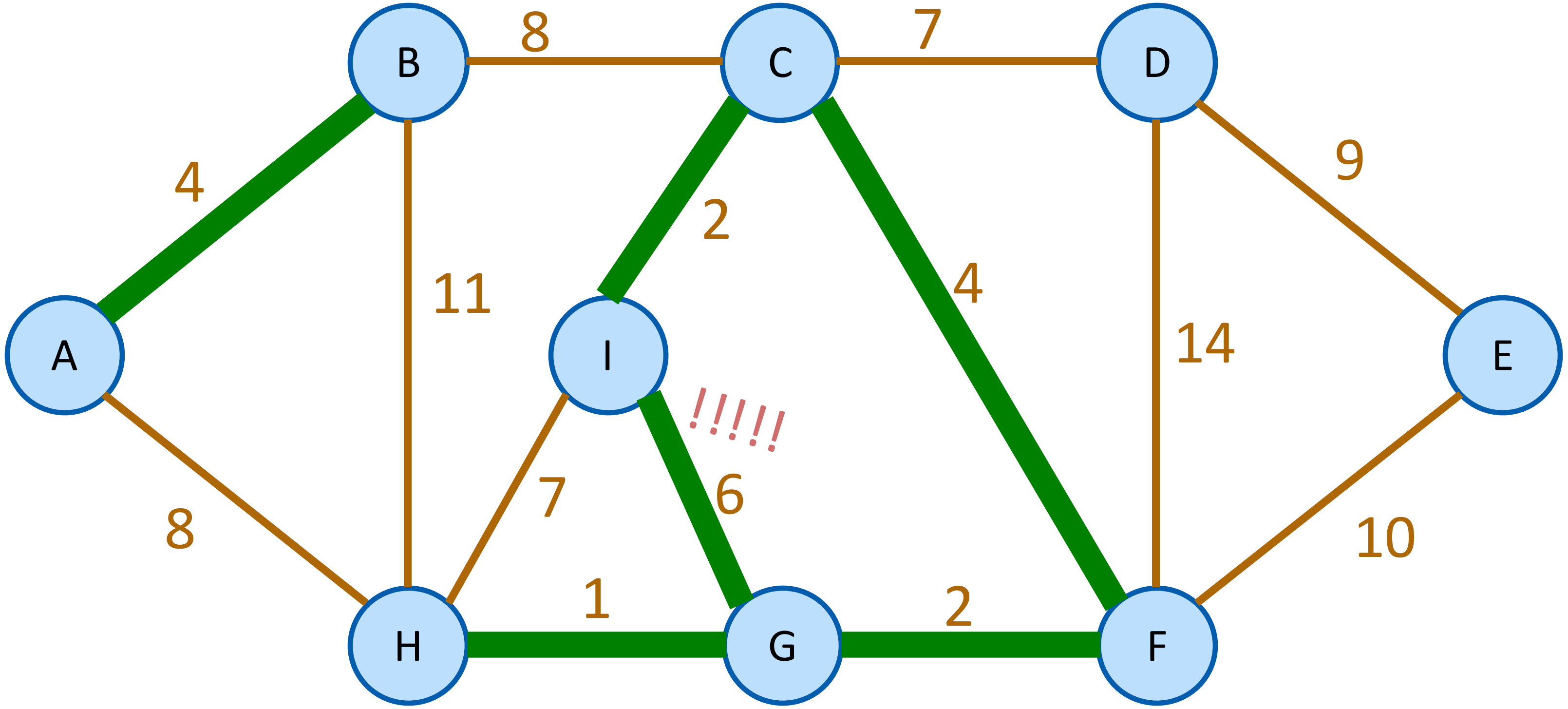
what if we just always take the cheapest edge?
whether or not it's connected to what we have so far?



Thoughts

香港科技大学(广州)
THE HONG KONG
UNIVERSITY OF SCIENCE AND
TECHNOLOGY (GUANGZHOU)

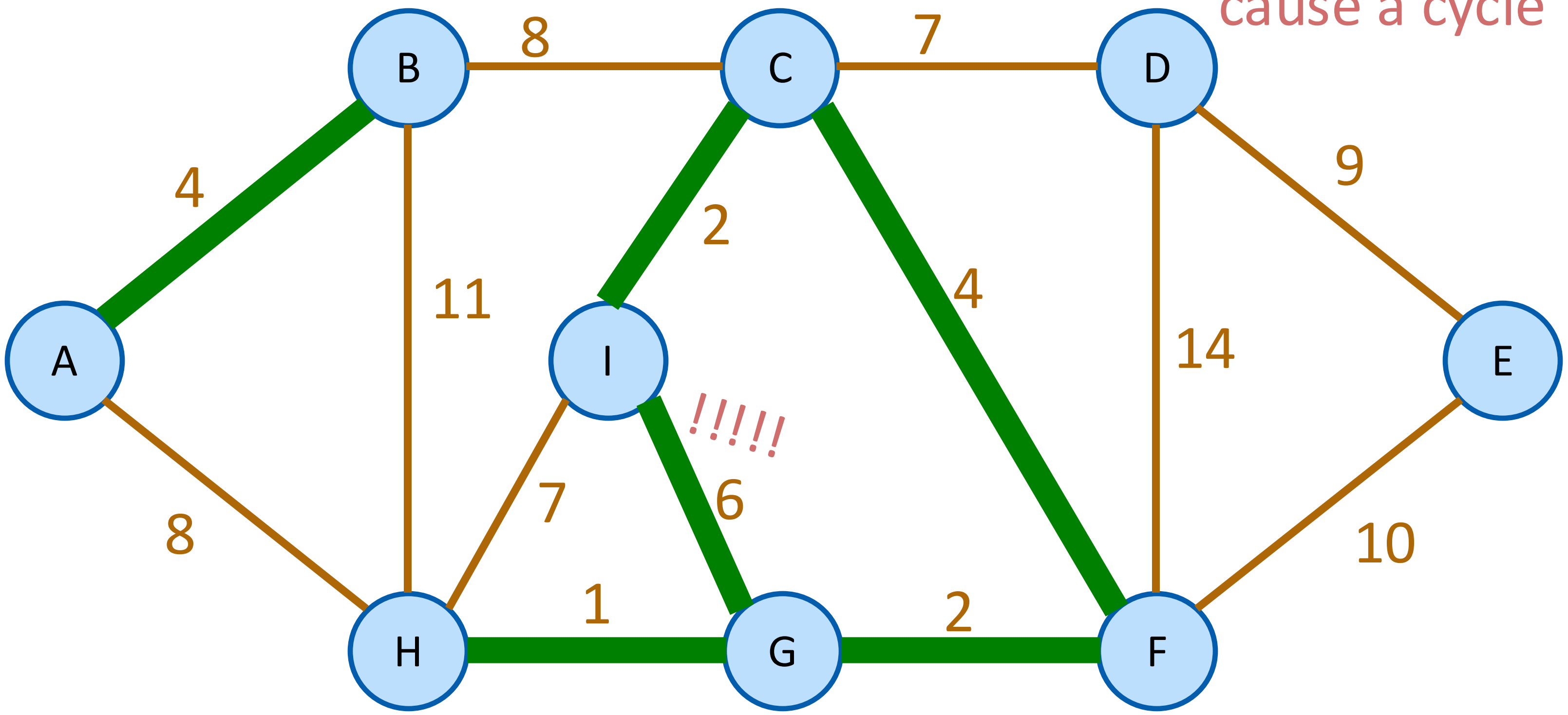
what if we just always take the cheapest edge?
whether or not it's connected to what we have so far?



Thoughts

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

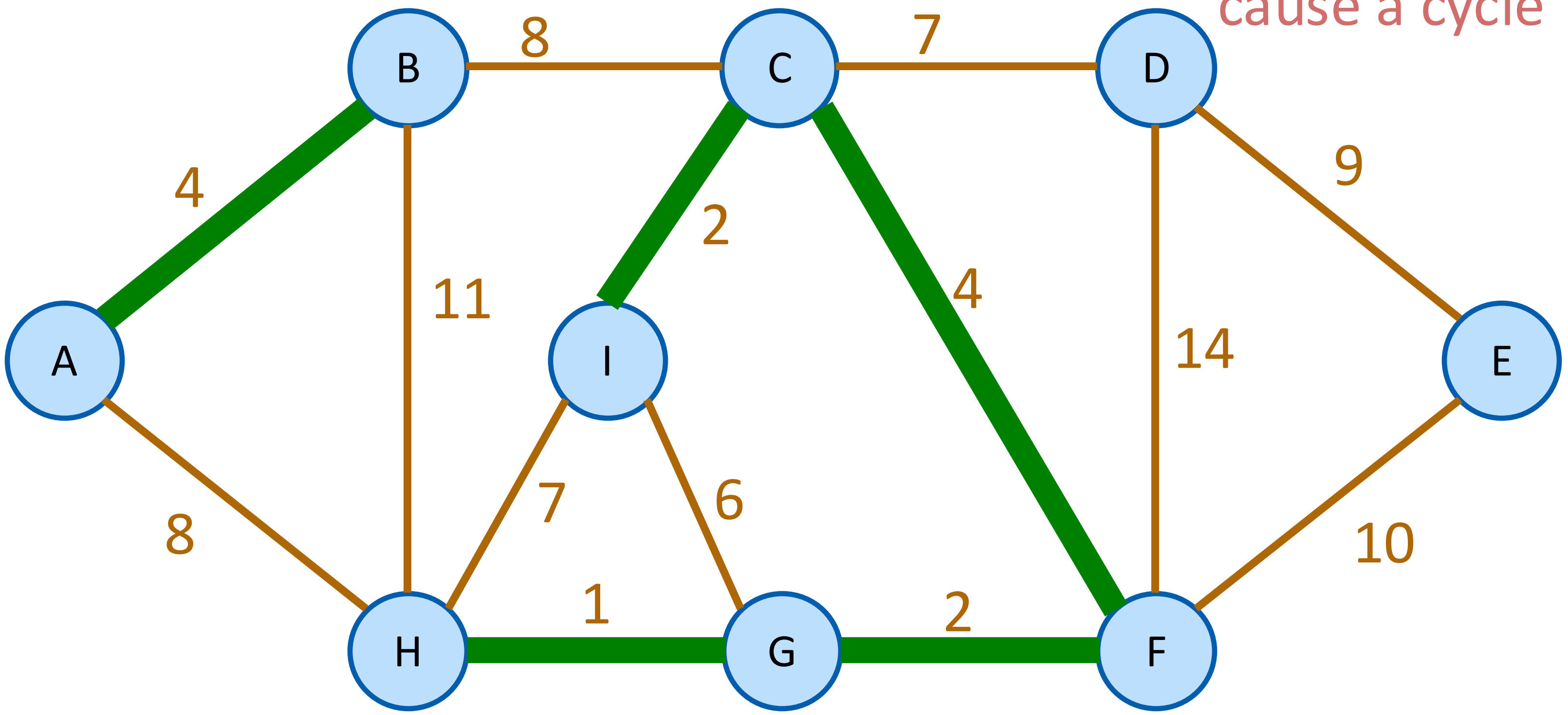
That won't cause a cycle



Thoughts

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

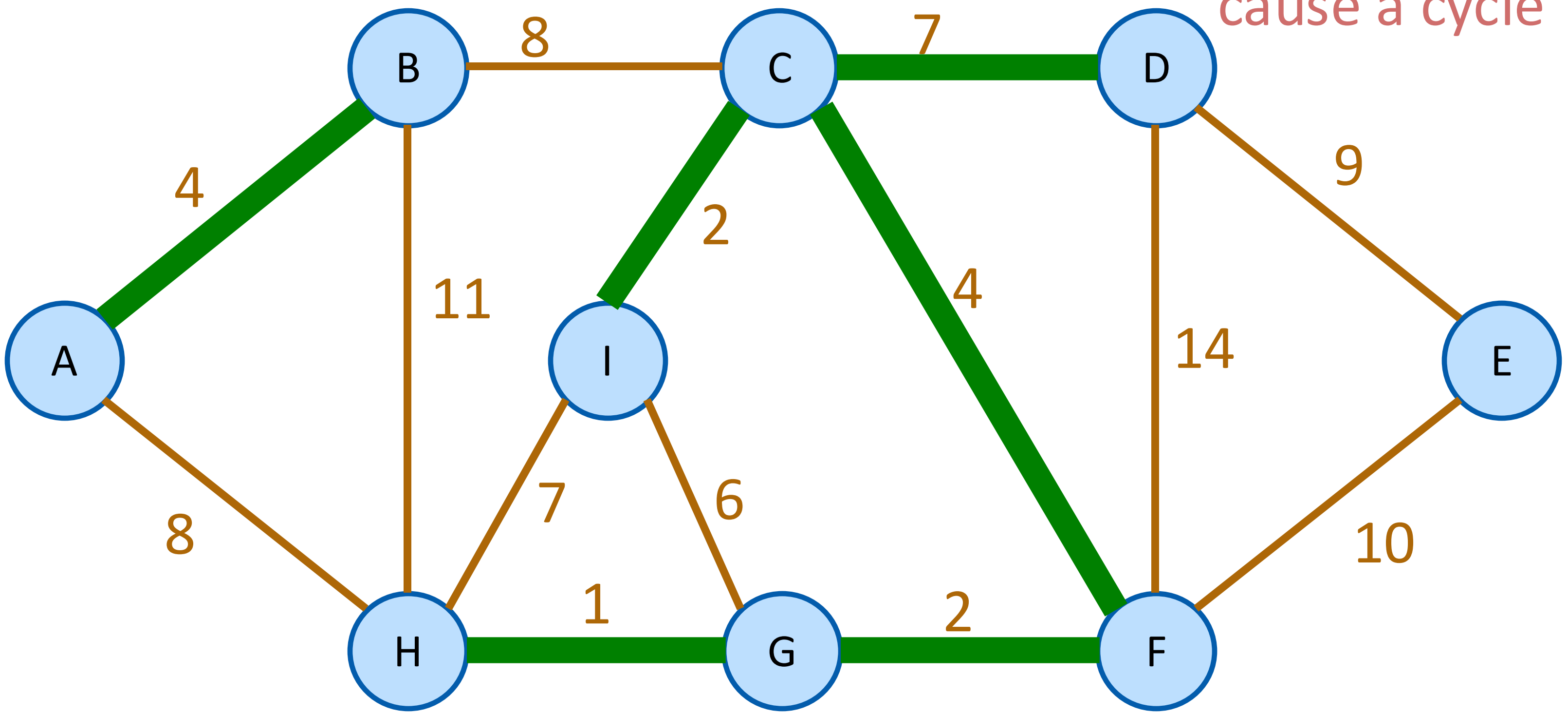
That won't cause a cycle



Thoughts

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

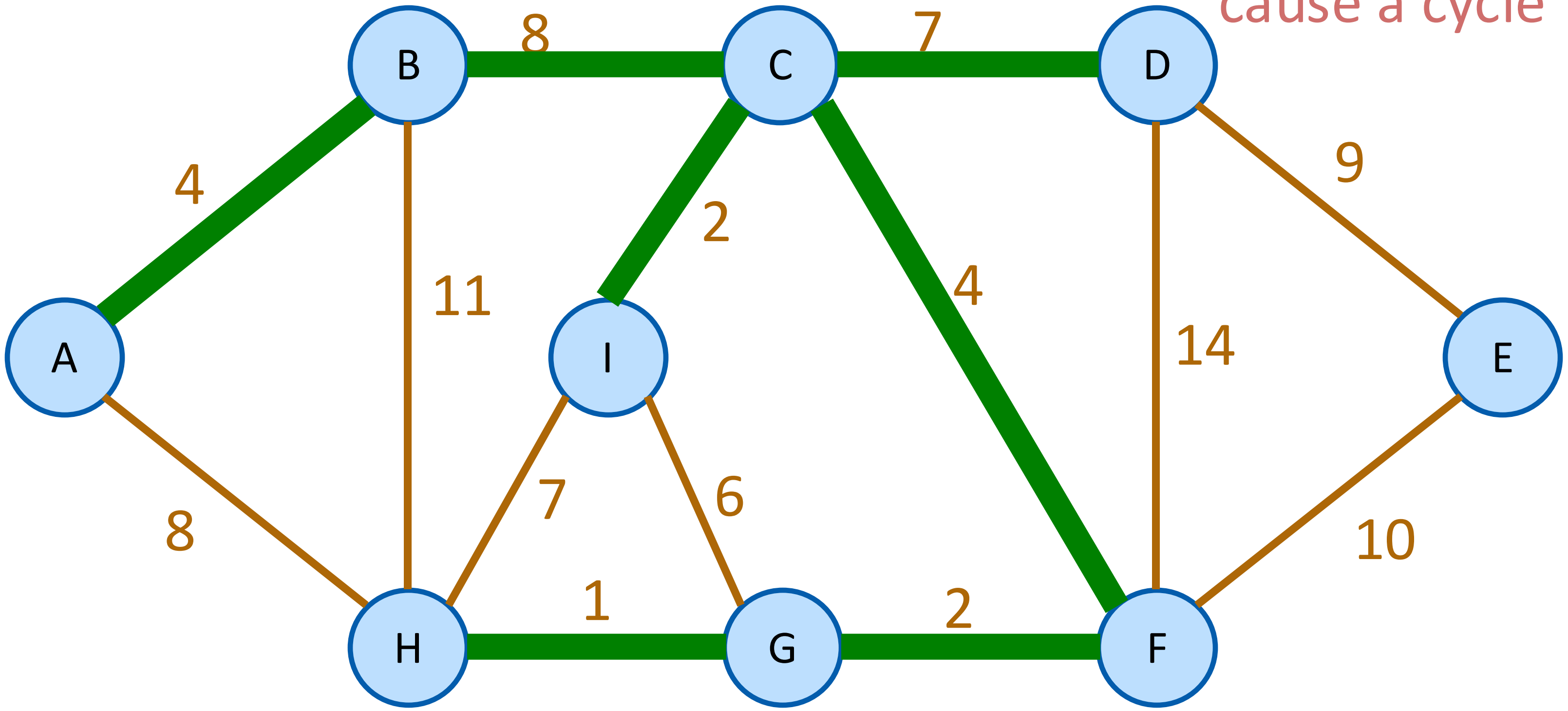
That won't cause a cycle



Thoughts

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

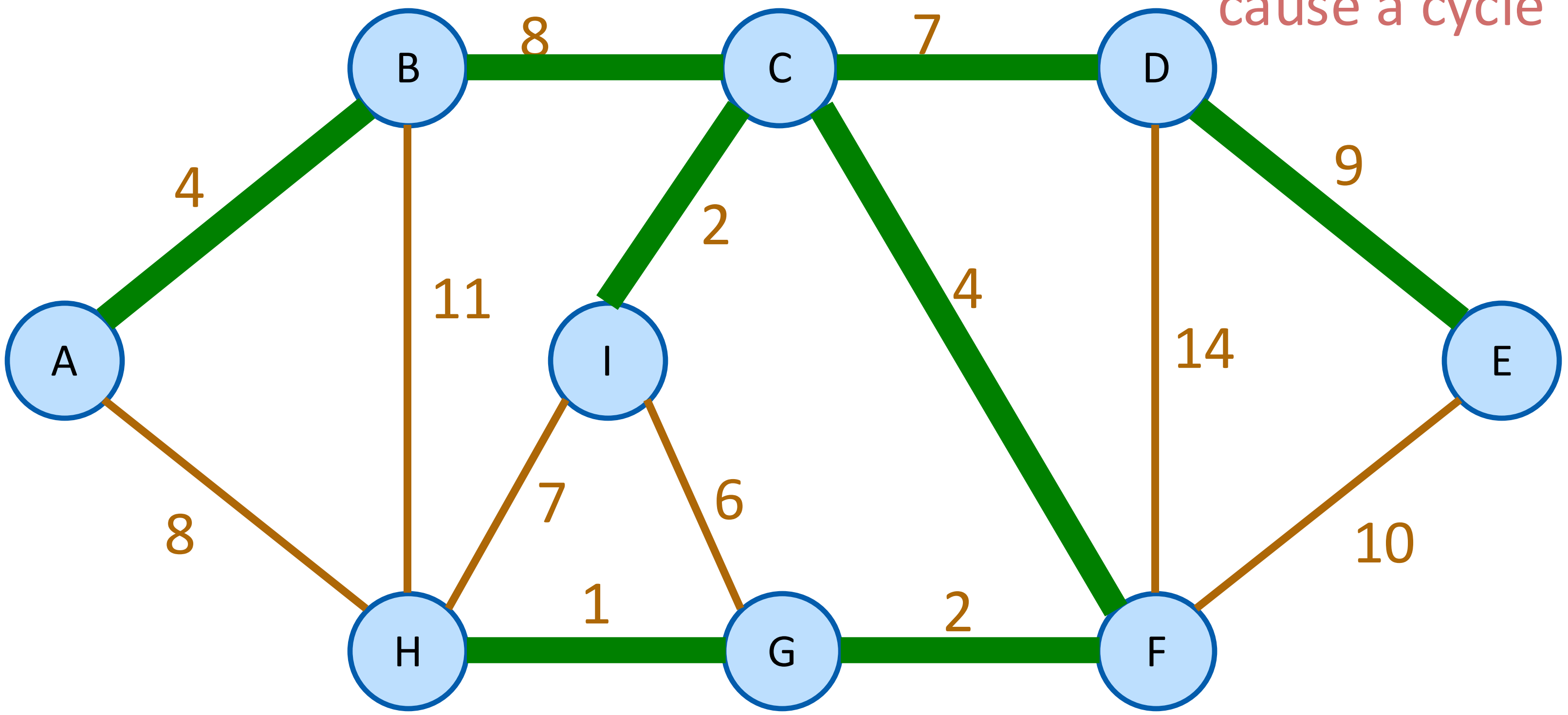
That won't cause a cycle



Thoughts

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



Kruskal’s Algorithm

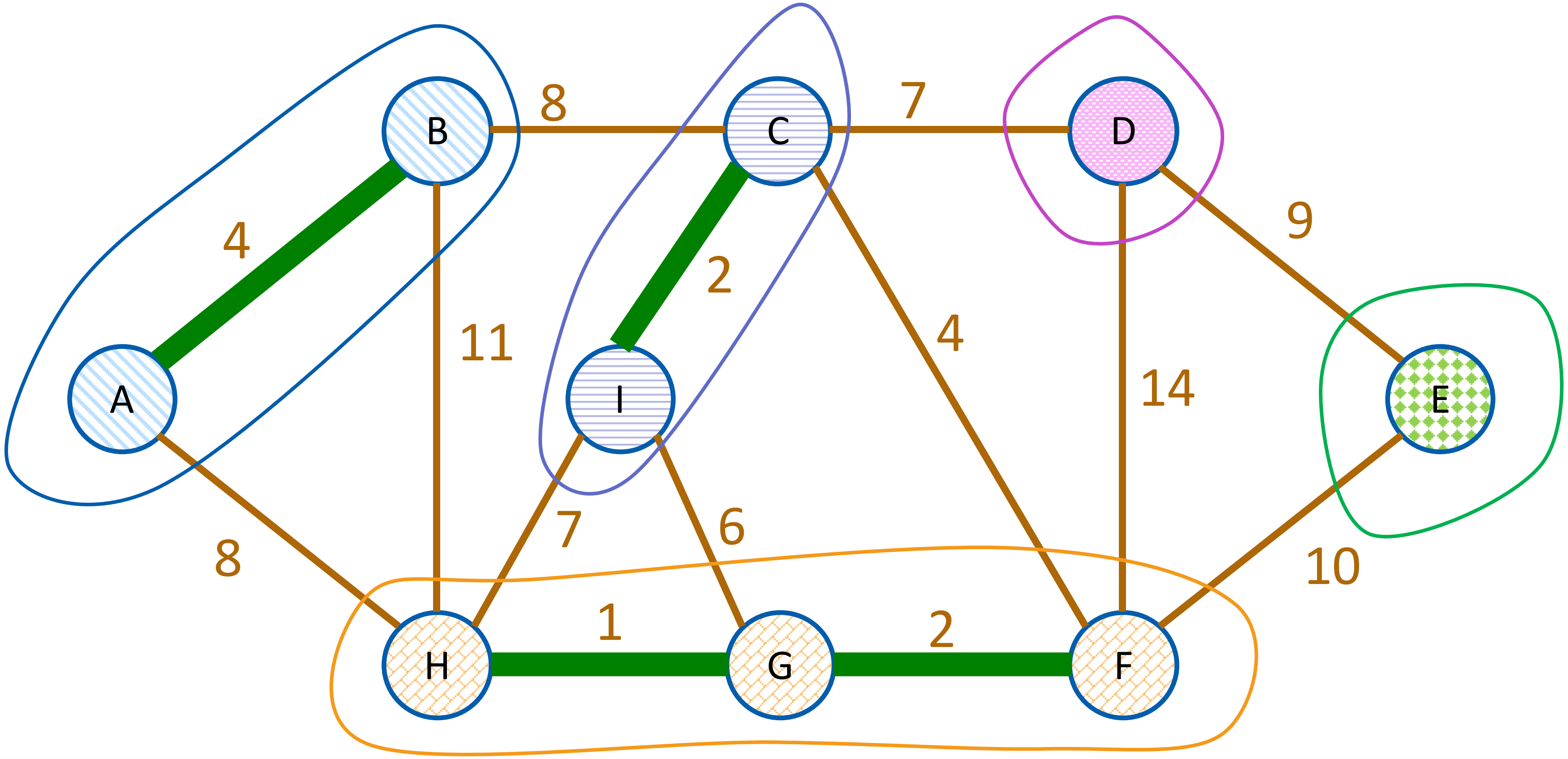
- **slowKruskal**(G = (V,E)):
  - Sort the edges in E by non-decreasing weight.
  - MST = {}
  - **for** e in E (in sorted order):
    - **if** adding e to MST won’t cause a cycle:
      - add e to MST.
  - **return** MST

m iterations through this loop

How do we check this?

Kruskal's Algorithm

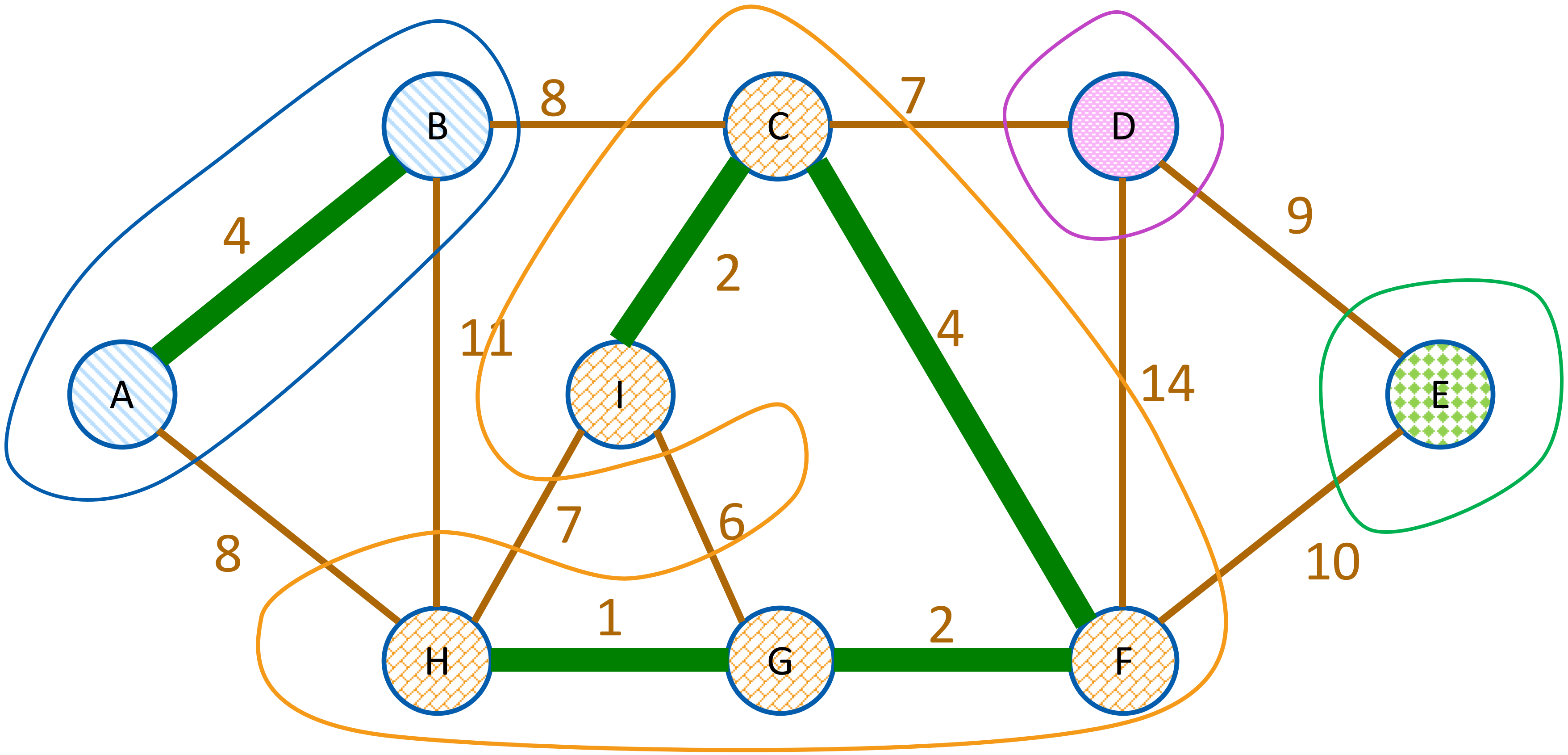
At each step of Kruskal's, we are maintaining a forest.



Kruskal's Algorithm

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

When we add an edge, we merge two trees:

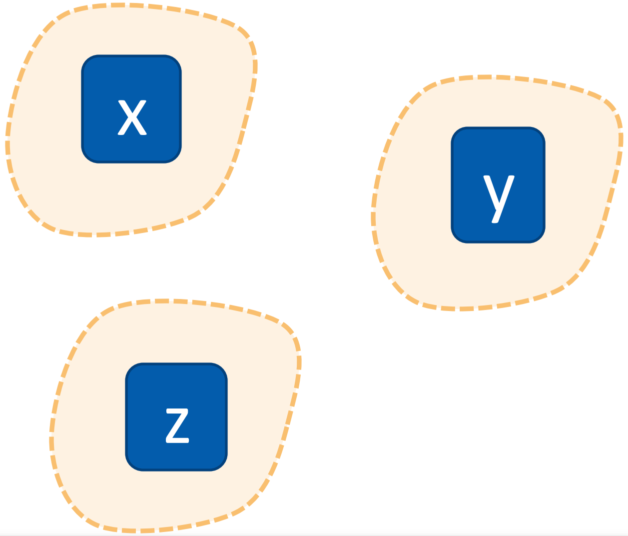


Union-find data structure

- 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)
```



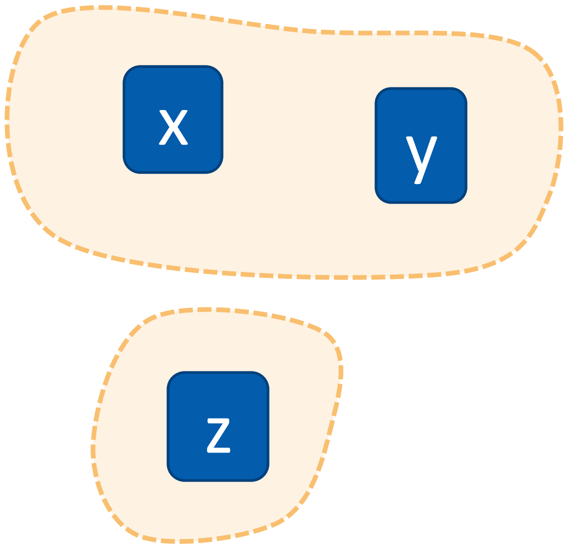
Kruskal’s Algorithm

Union-find 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)
```



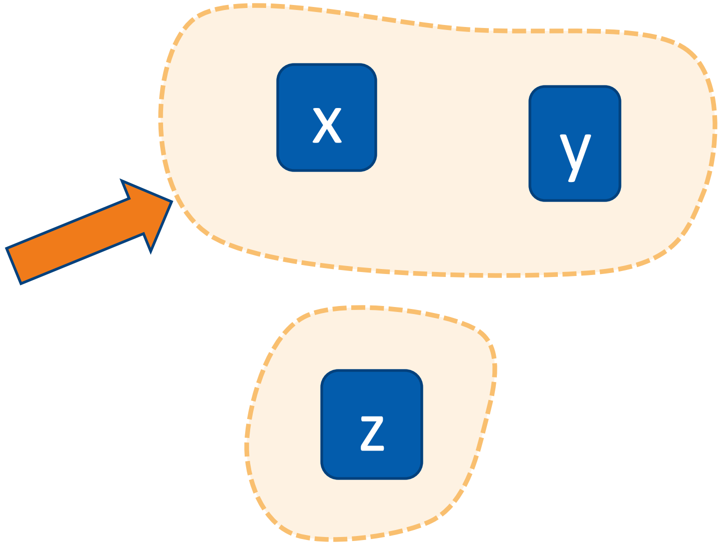
Kruskal’s Algorithm

Union-find 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's Algorithm

- **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

Kruskal's Algorithm

Running time

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

Kruskal's Algorithm

Does it work?

Leave for your assignment.

Comparison

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

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

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

Can we do better?

- Karger-Klein-Tarjan 1995:
  - O(m) time randomized algorithm
- Chazelle 2000:
  - O(m$\cdot$ $\alpha(n)$) time deterministic algorithm
- Pettie-Ramachandran 2002:
  - O$\left(\begin{array}{c}\text{The optimal number of comparisons} \\ \text{you need to solve the problem,} \\ \text{whatever that is...}\end{array}\right)$ time deterministic algorithm

The End