COMP - 285

Analysis of Algorithms

Welcome to COMP 285

Lecture 17: Minimum Spanning Trees

Lecturer: Chris Lucas (cflucas@ncat.edu)

HW5 due!

Tonight @ 11:59PM ET

HW6 released by EoD!

Due 11/03 @ 11:59PM ET

Final Exam!

Tuesday 12/06 from 2:00pm-4:00pm

Career Office Hours

- Denzel from Meta (sign ups here)
- Resume, general advice, behavioral interview practice, Meta, etc.

Netflix Opportunity!

- Fill out this form (link)
- Check “Meta Classroom”

Quiz!

www.comp285-fall22.ml or Blackboard

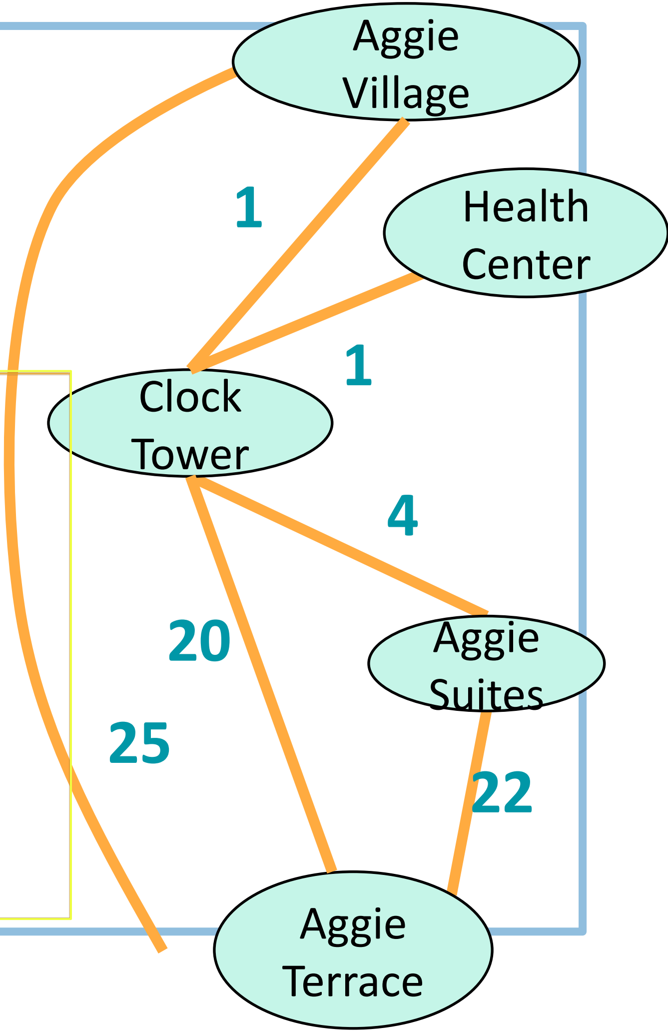
Recall where we ended last lecture…

Dijkstra's algorithm

- Finds shortest paths from Aggie Village to everywhere else.

Dijkstra(G,s):

- Set all vertices to **not-sure**
- $d[v] = \infty$ for all v in V
- $d[s] = 0$
- **While** there are **not-sure** nodes:
  - Pick the **not-sure** node u with the smallest estimate **d[u].**
  - **For** v in u.neighbors:
    - $d[v] \leftarrow \min(\ d[v]\ ,\ d[u] + \text{edgeWeight}(u,v))$
  - Mark u as **sure**.
- Now $d(s, v) = d[v]$



We need a data structure that…

- Stores unsure vertices v
- … And keeps track of d[v]
- Can find u with minimum d[u]
  - `findMin()`
- Can remove that u
  - `removeMin(u)`
- Can update (decrease) d[v]
  - `updateKey(v,d)`

Just the inner loop:

- Pick the **not-sure** node u with the smallest estimate **d[u].**
- Update all u's neighbors v:
  - d[v] ← min(d[v] , d[u] + edgeWeight(u,v))
- Mark u as **sure**.

We need a data structure that...

- Stores unsure vertices v
- … And keeps track of d[v]
- Can find u with minimum d[u]
    - `findMin()`
- Can remove that u
    - `removeMin(u)`
- Can update (decrease) d[v]
    - `updateKey(v,d)`

Just the inner loop:

- Pick the **not-sure** node u with the smallest estimate **d[u].**
- Update all u's neighbors v:
    - $d[v] \leftarrow \min(d[v] , d[u] + \text{edgeWeight}(u,v))$
- Mark u as **sure**.

= V * (findMin + removeMin) + E * updateKey

If we use an array...

- findMin = O(V)
- removeMin = O(V)
- updateKey = O(1)

Running time of Dijkstra
= O(V * (findMin + removeMin) + E * updateKey)
= O(V * (V + V) + E * 1)
= $O(V^2 + E)$

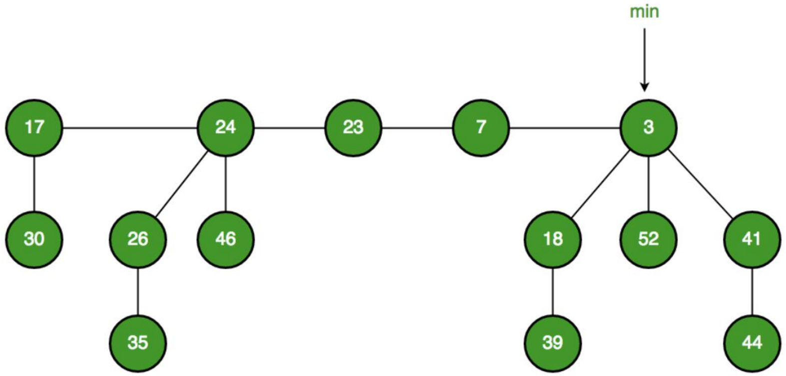
If we use a (balanced) BST...

- findMin = O(log(V))
- removeMin = O(log(V))
- updateKey = O(log(V))

Running time of Dijkstra
= O(V * (findMin + removeMin) + E * updateKey)
= O(V * (log(V) + log(V)) + E * log(V))
= O(Vlog(V)+ Elog(V))
= O((V+E)log(V))

Say we use a fibonacci heap…

- findMin = O(1)
- removeMin = O(log(V)*)
- updateKey = O(1)



Running time of Dijkstra

= O(V * (findMin + removeMin) + E * updateKey)

= O(V * (1 + log(V)) + E * 1)

= O(Vlog(V)+ E)

*amortized time: any sequence of d `removeMin` calls takes time at most O(dlog(n)). But a few of the d may take longer than O(log(n)) and some may take less time..

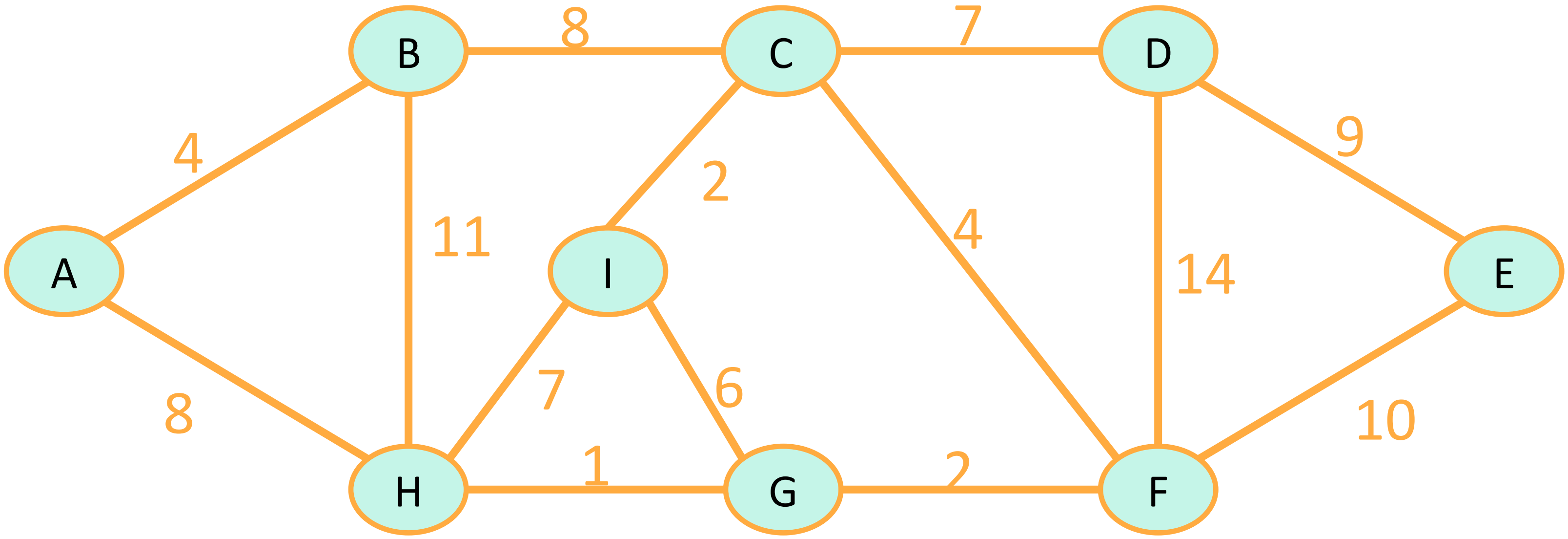
Big Questions!

- What's a spanning tree/minimum spanning tree?
- How do we find a minimum spanning tree?
- How do we find a minimum spanning tree (pt. 2)?

Big Questions!

- What's a spanning tree/minimum spanning tree?
- How do we find a minimum spanning tree?
- How do we find a minimum spanning tree (pt. 2)?

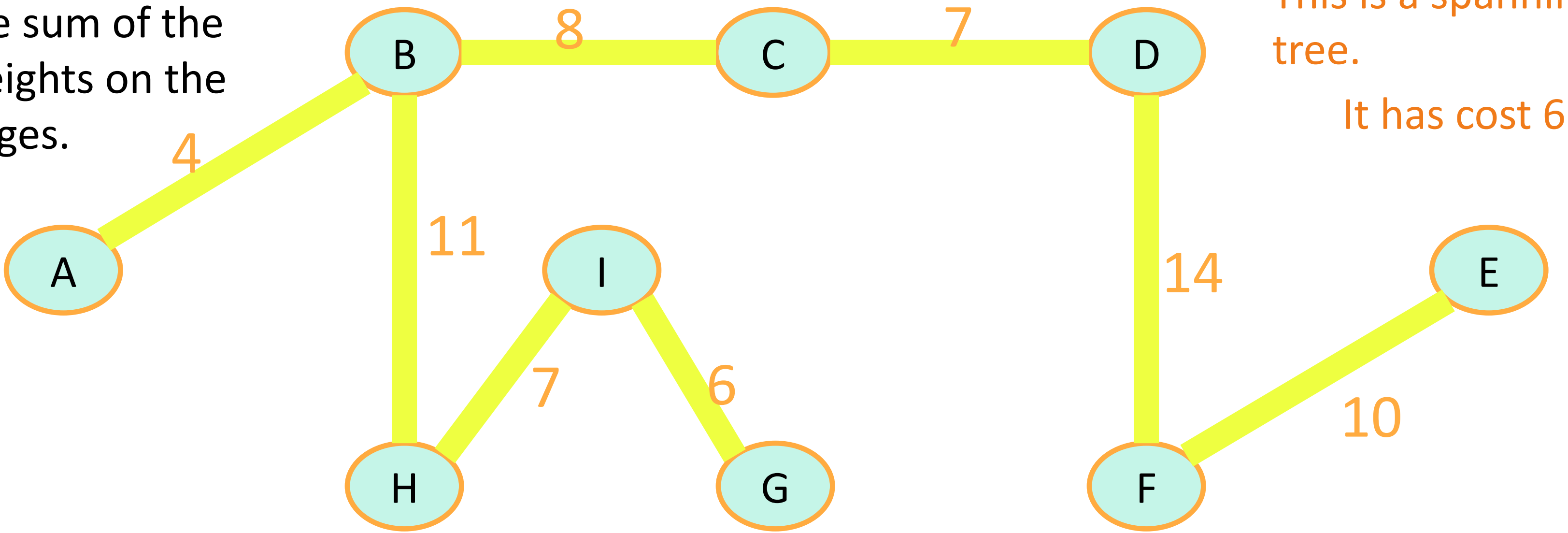
Say we have an undirected weighted graph...



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

Say we have an undirected weighted graph...

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

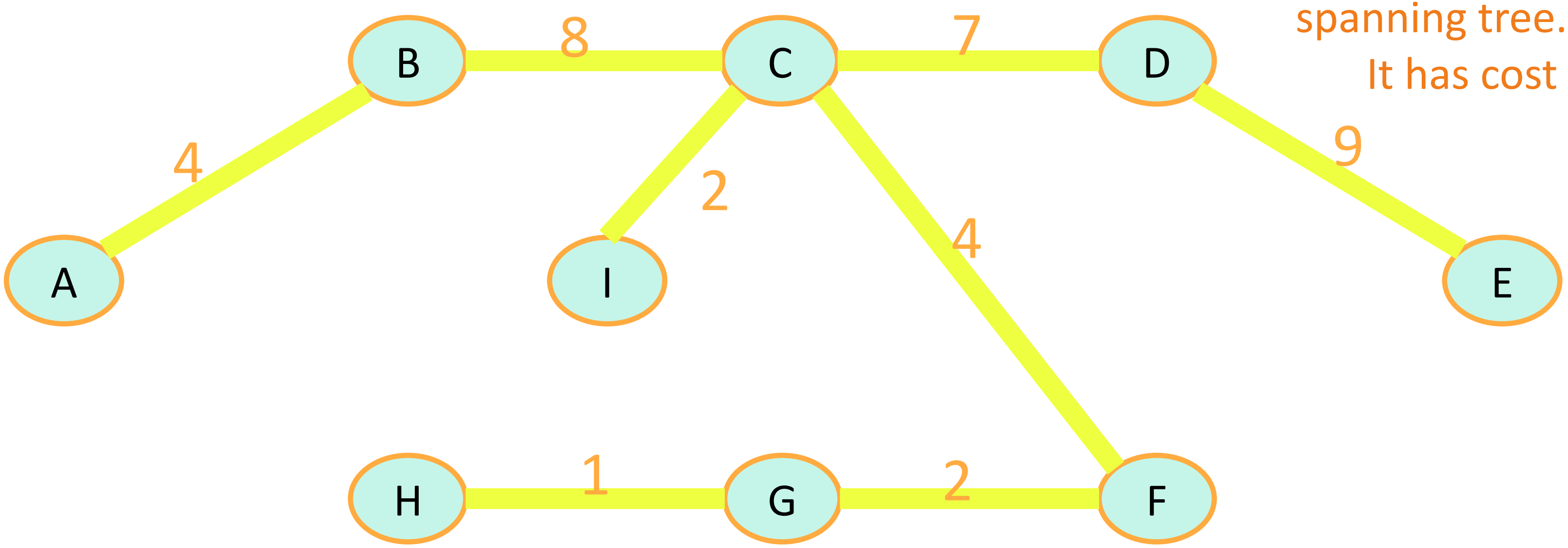


This is a spanning tree.

It has cost 67

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

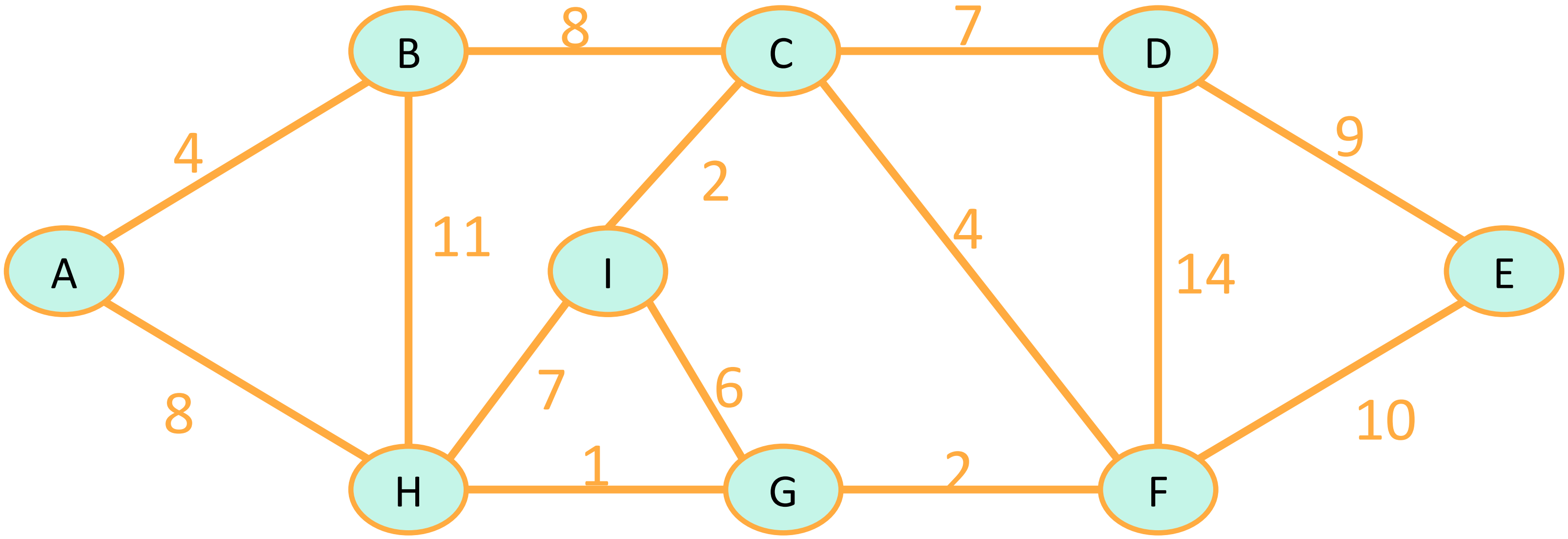
Say we have an undirected weighted graph…



This is also a spanning tree.
It has cost 37

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

What’s a Minimum Spanning Tree (MST)?



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

Kahoot!

www.kahoot.it, Code: XXX YYYY

Enter your @aggies.ncat email

What’s a Minimum Spanning Tree (MST)?

- True or False? There can only be **one** Spanning Tree in a connected graph G.
  - False
- True or False? There can only be **one** MST in a connected graph G.
  - False
- True or False? In some cases, you may have an MST with >|V| - 1 edges.
  - False

Why Minimum Spanning Trees (MSTs)?

- Network Design
  - Connecting cities with roads/electricity/telephone/…
- Cluster Analysis
  - eg, genetic distance
- Image Processing
  - eg, image segmentation
- Useful primitive
  - for other graph algs

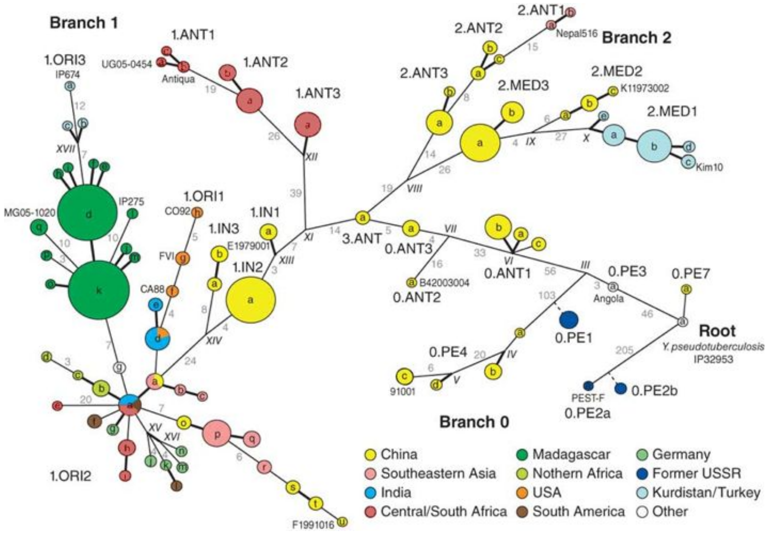


Figure 2: Fully parsimonious minimal spanning tree of 933 SNPs for 282 isolates of *Y. pestis* colored by location.

Morelli et al. Nature genetics 2010

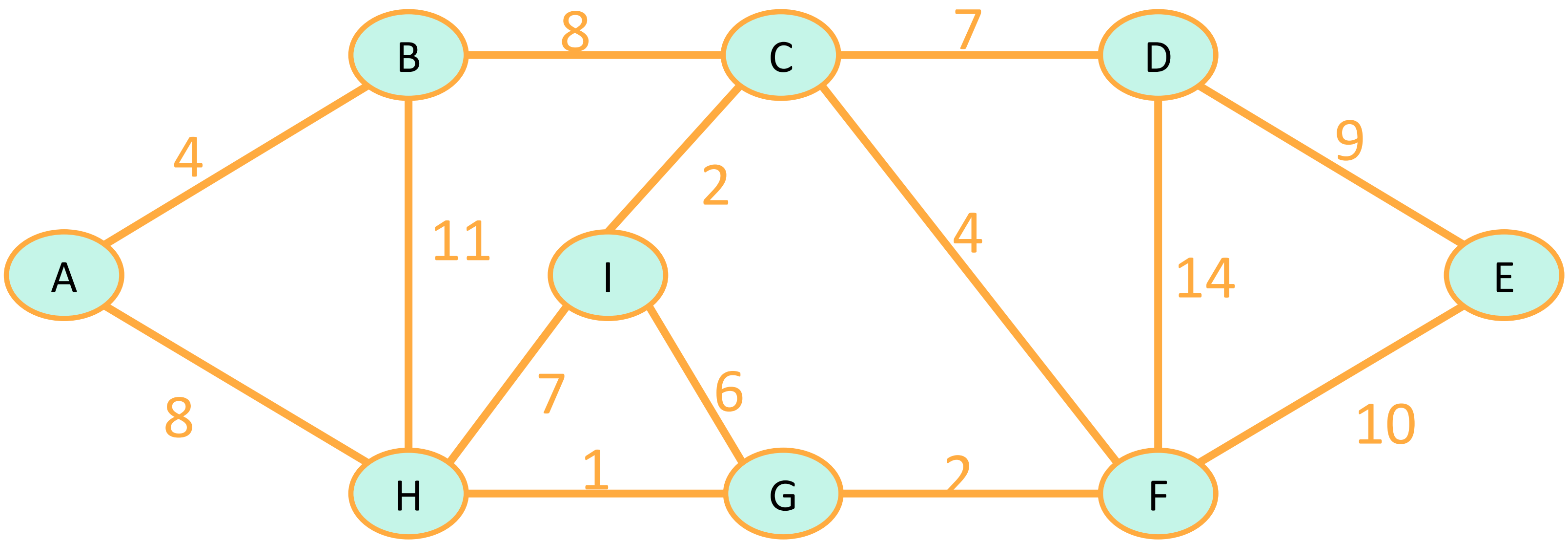
Big Questions!

- What's a spanning tree/minimum spanning tree?
- How do we find a minimum spanning tree?
- How do we find a minimum spanning tree (pt. 2)?

How to find an MST?

- Minimum Spanning Tree (MST) Problem
  - Input: weighted, undirected, connected Graph $G = (V, E)$
  - Output: A Tree $T = (V, E')$, with $E' \subseteq E$ that minimizes the edge weight sum
- This is a formal way of saying "An MST is a Spanning Tree with a connected, weighted graph that has the least total cost when you sum all the edges."
- Today, we will explore two (greedy) algorithms!

Brainstorm some algorithms...

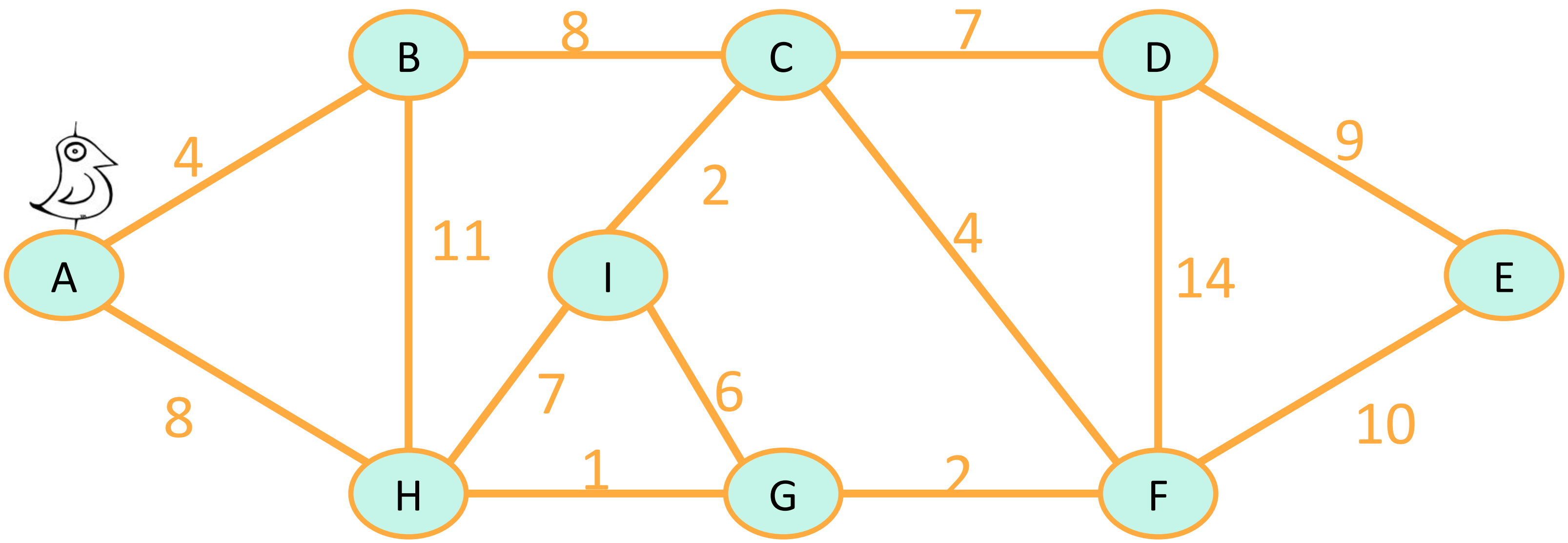


Back to MSTs

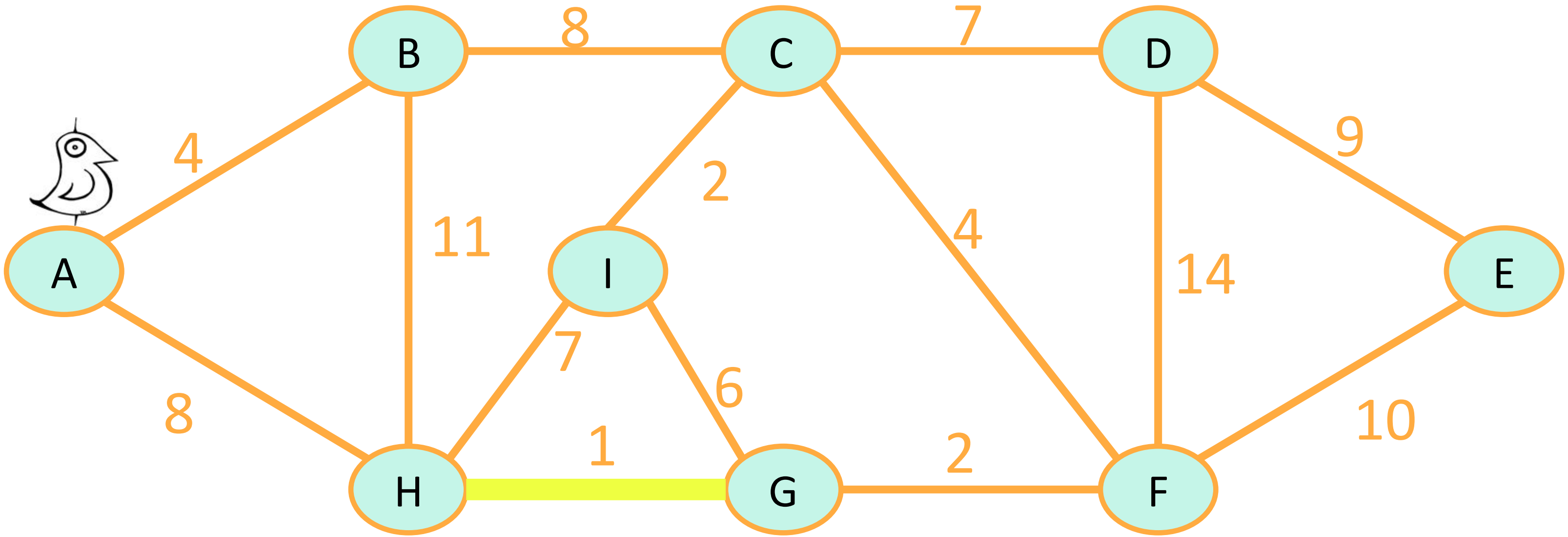
The strategy:

1. Make a series of choices, adding edges to the tree.
2. Show that each edge we add is **safe to add**:
    - We do not create cycles
3. **Repeat** until we have an MST.

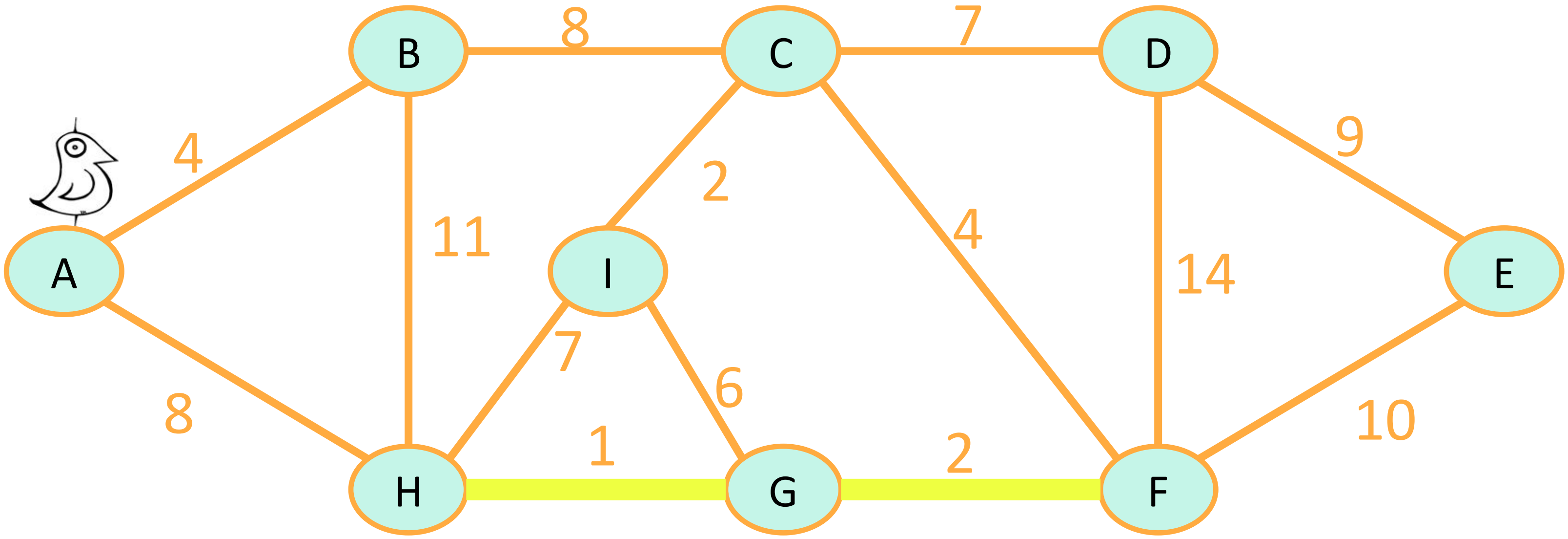
Idea 1: Start growing a tree, (greedily) add the shortest edge we can to grow the tree.



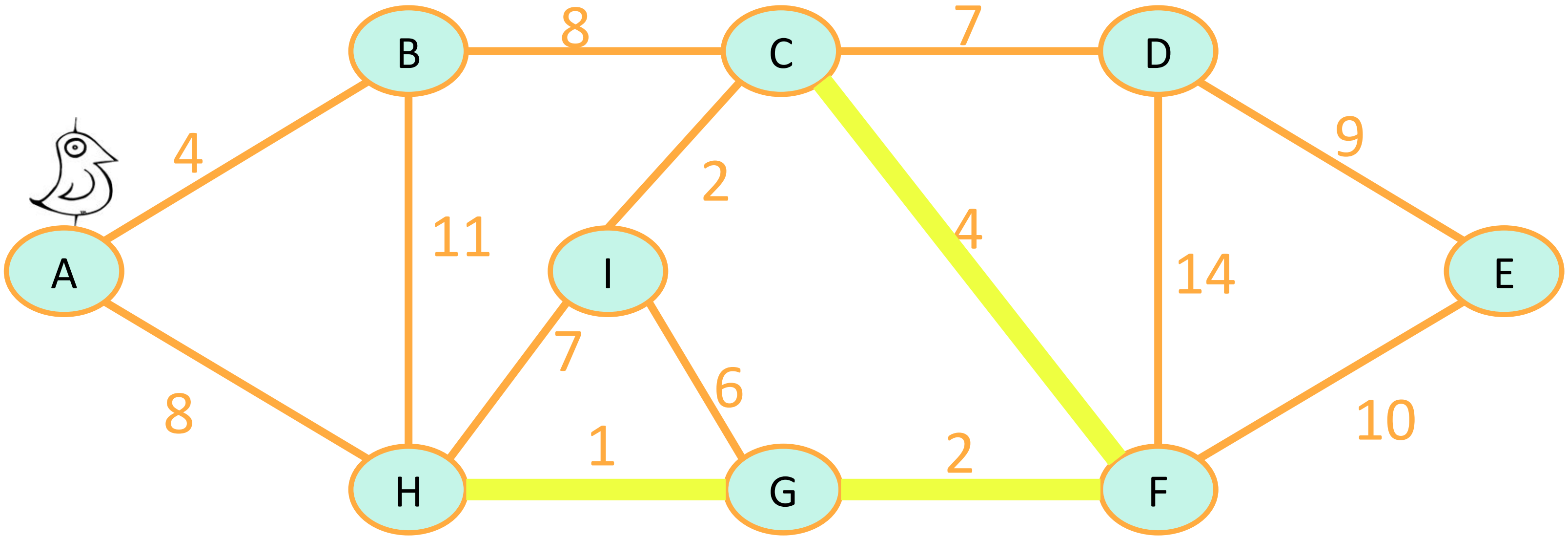
Idea 1: Start growing a tree, (greedily) add the shortest edge we can to grow the tree.



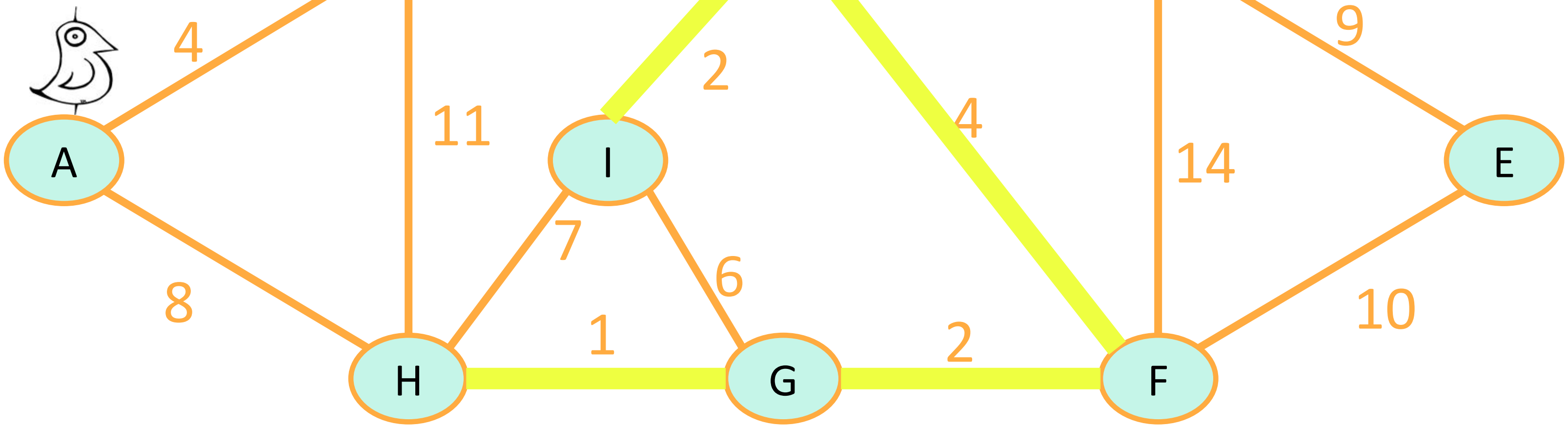
Idea 1: Start growing a tree, (greedily) add the shortest edge we can to grow the tree.



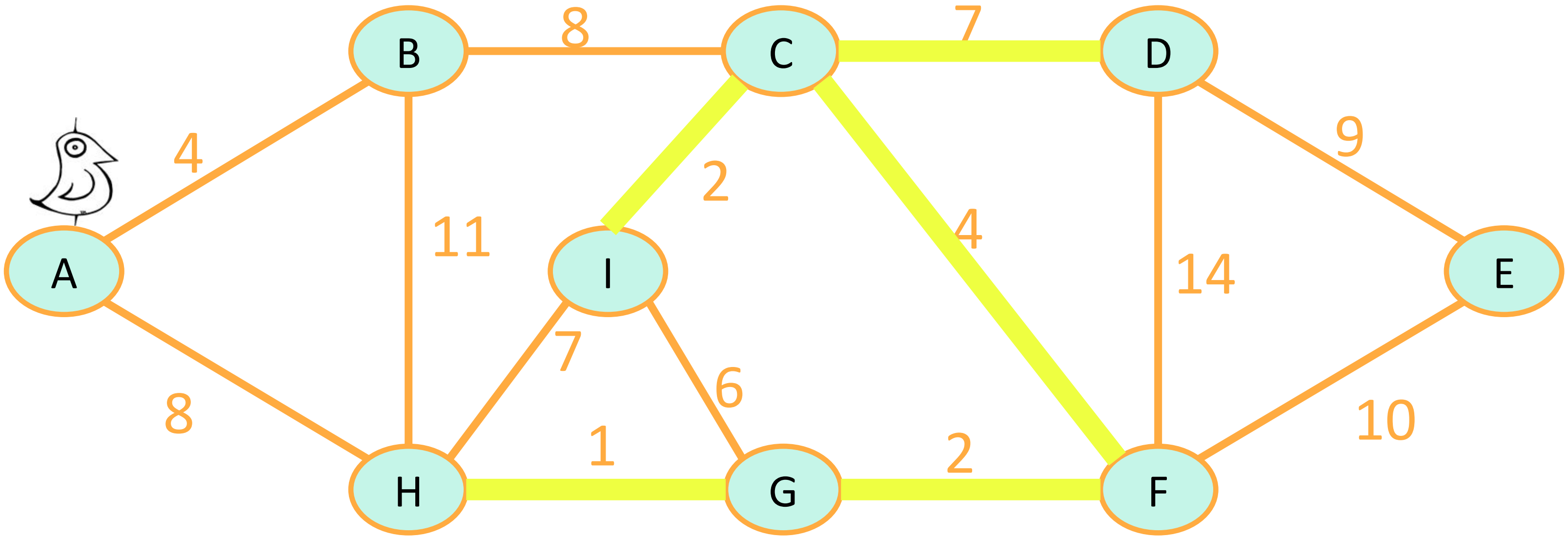
Idea 1: Start growing a tree, (greedily) add the shortest edge we can to grow the tree.



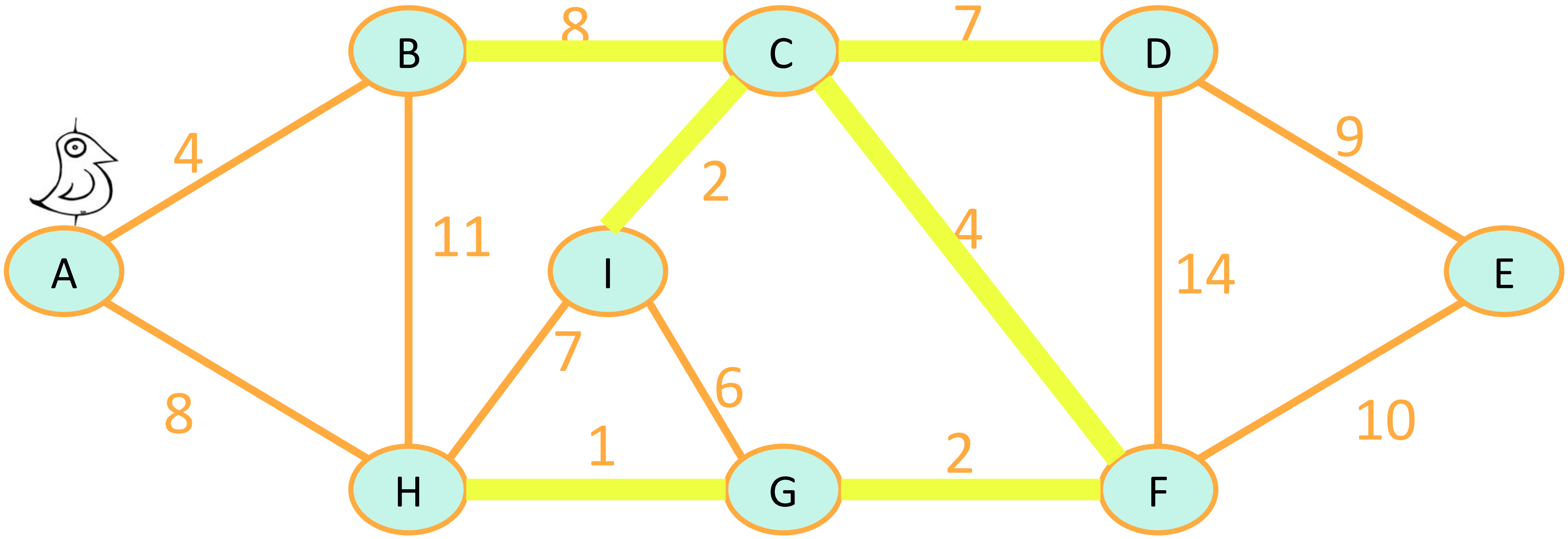
Idea 1: Start growing a tree, (greedily) add the shortest edge we can to grow the tree.



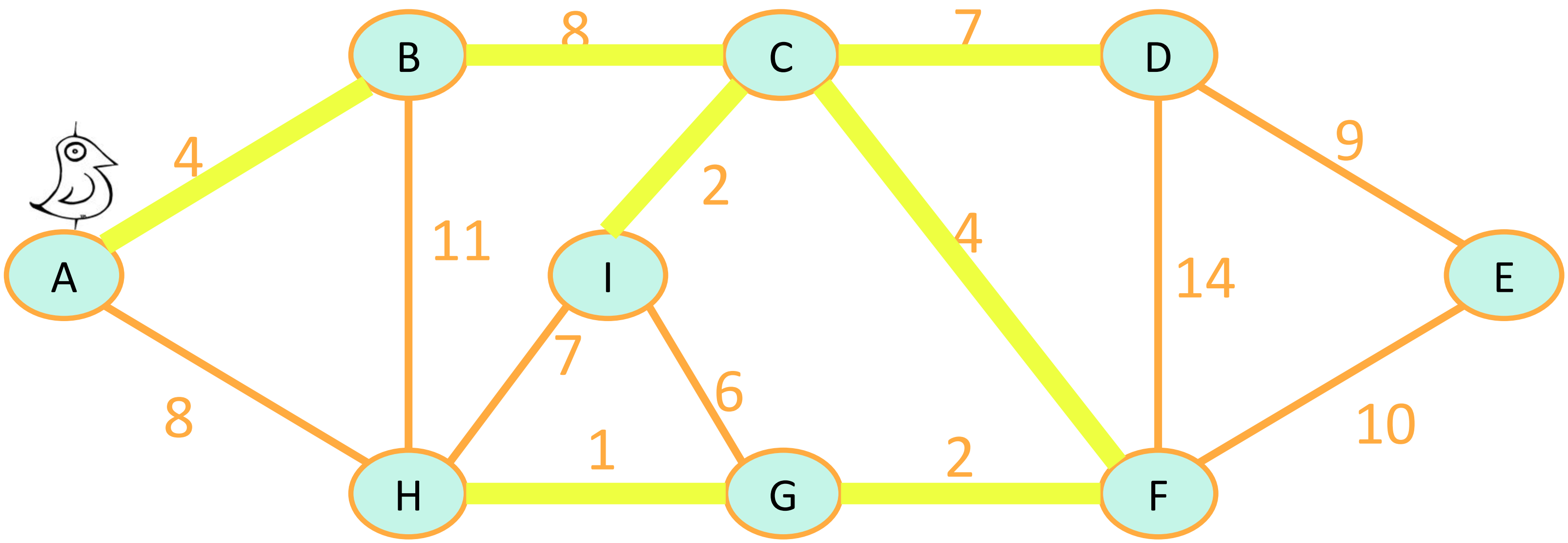
Idea 1: Start growing a tree, (greedily) add the shortest edge we can to grow the tree.



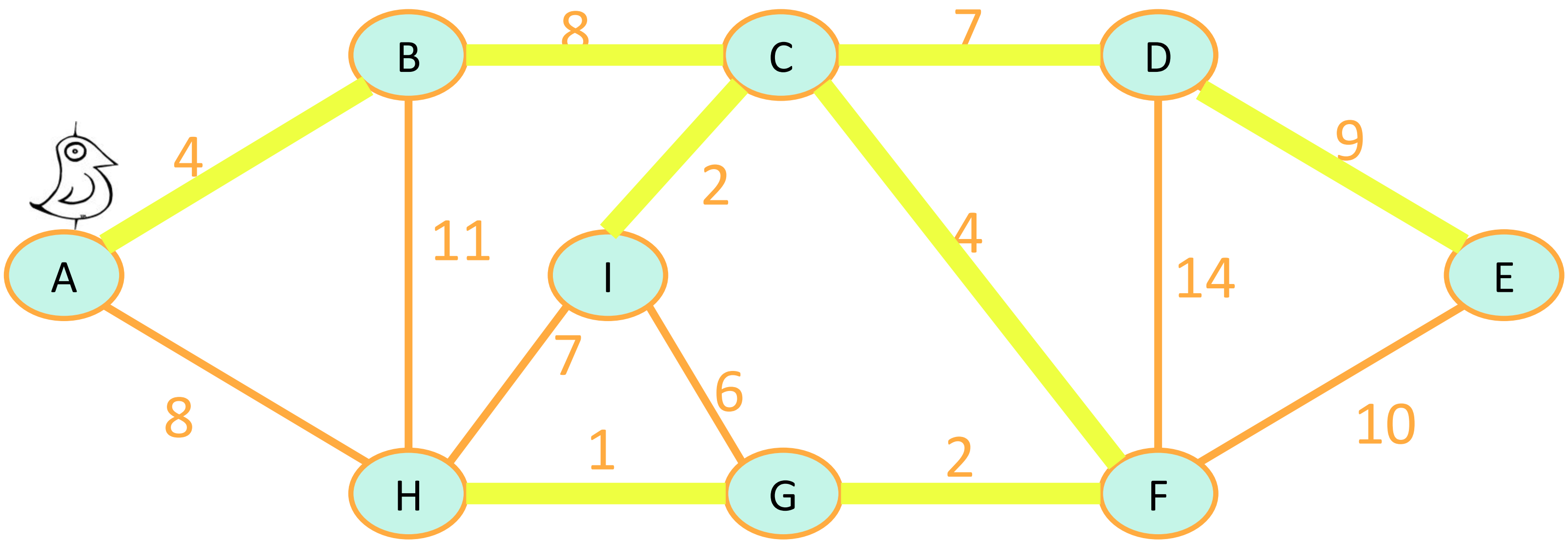
Idea 1: Start growing a tree, (greedily) add the shortest edge we can to grow the tree.



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(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

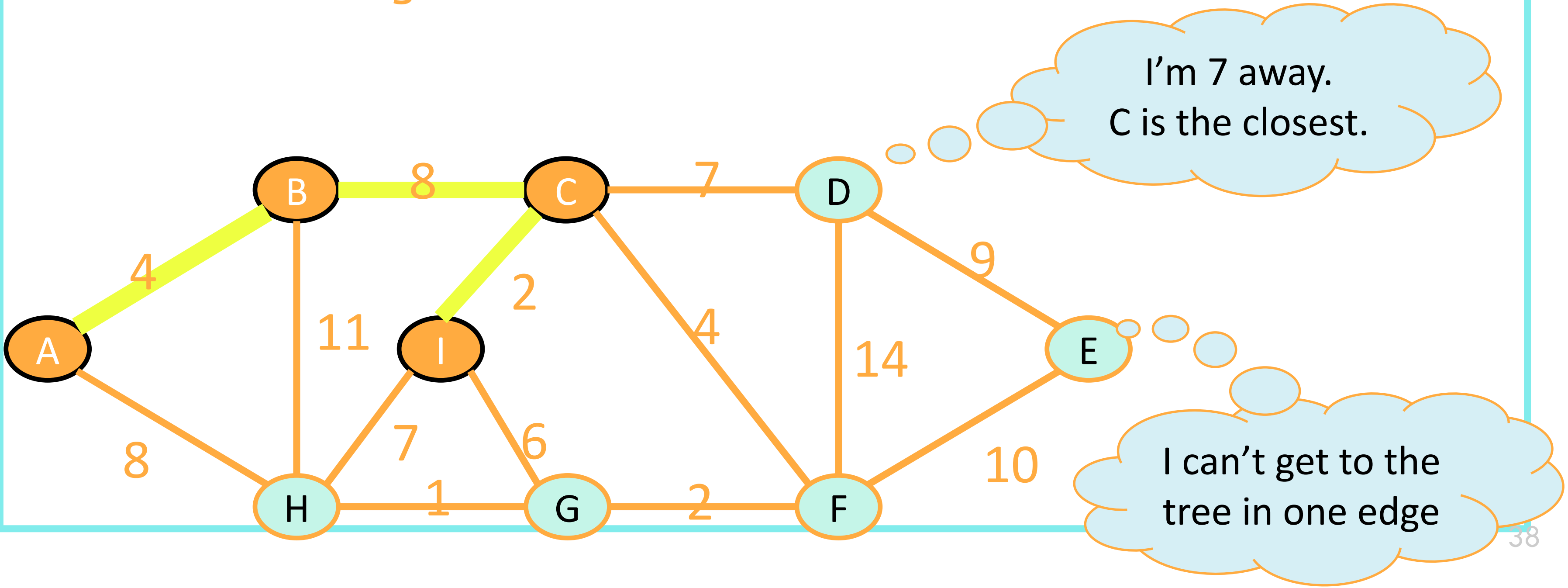
At most V iterations of this while loop.

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

How do we actually implement this?

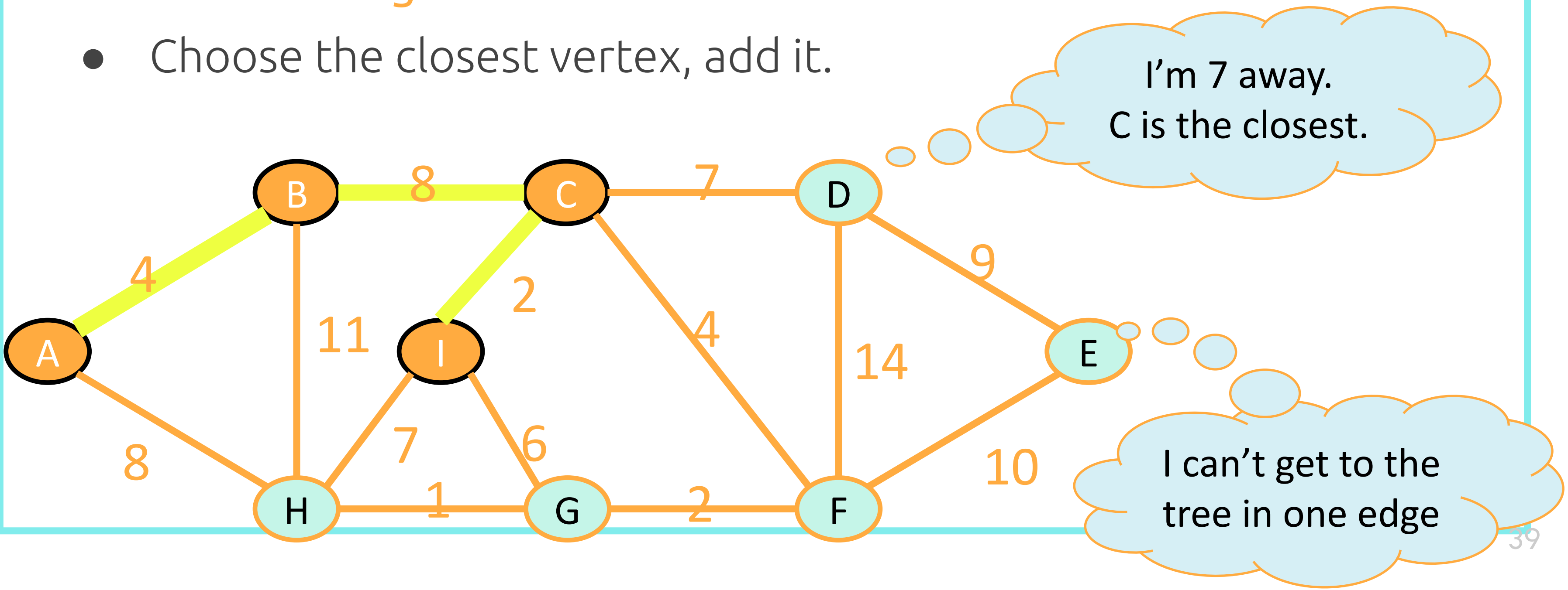
- Each vertex keeps:
  - the **distance** from itself to the **growing spanning tree**
  - **how to get there.**

if you can get there in one edge.



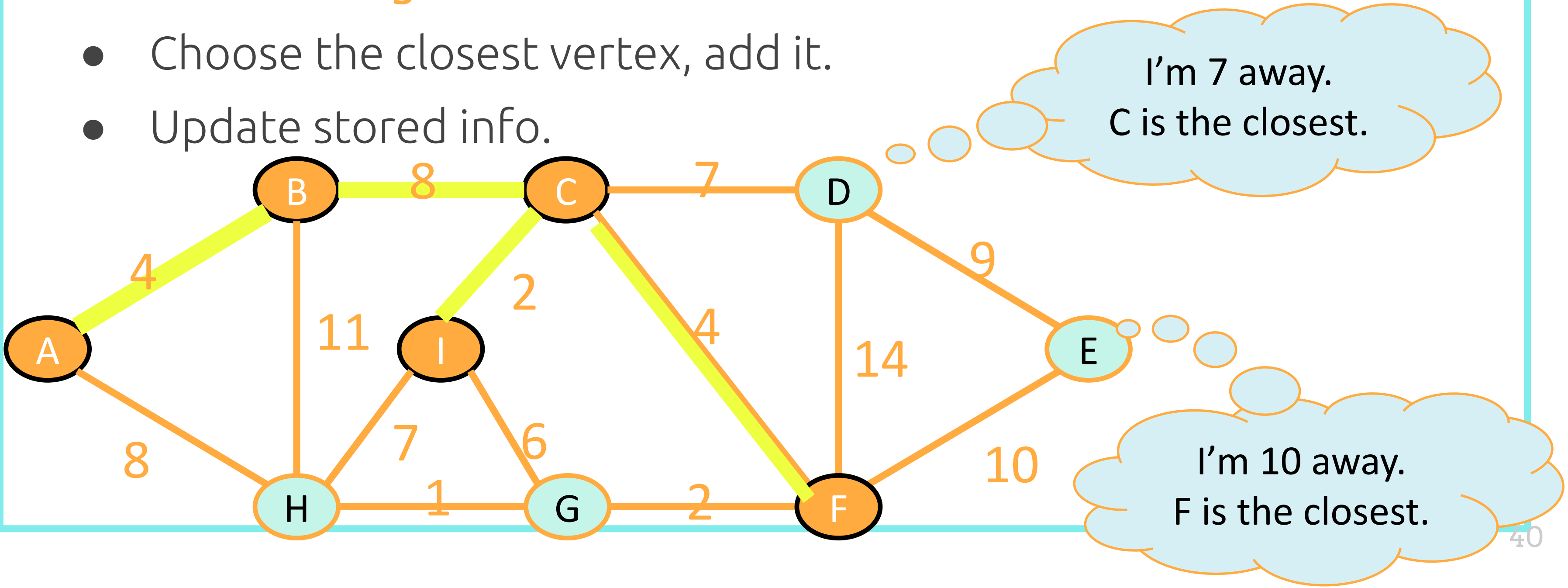
How do we actually implement this?

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



How do we actually implement this?

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



We’ve Discovered Prim’s Algorithm!

```
algorithm Prim’s
  Input: Weighted, Undirected, connected Graph G = (V, E) with edge weights w_e
  Output: A Tree T = (V, E'), with E' ⊆ E that minimizes the edge weight sum

  for all u ∈ V :
    cost(u) = ∞
    prev(u) = nil
  Pick any initial node u_0
  cost(u_0) = 0

  H = makequeue(V) (using cost-values as keys)
  while H is not empty:
    v = extractMin(H)
    for each z of v’s neighbors ({v, z} ∈ E):
      if cost(z) > w(v, z) and H.contains(z):
        cost(z) = w(v, z)
        prev(z) = v
        decreasekey(H, z)
```

Prim's Runtime

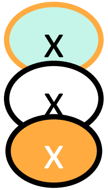
```
algorithm Prim's
  Input: Weighted, Undirected, connected Graph G = (V, E) with edge weights w_e
  Output: A Tree T = (V, E'), with E' ⊆ E that minimizes the edge weight sum

  for all u ∈ V :
    cost(u) = ∞
    prev(u) = nil
  Pick any initial node u_0
  cost(u_0) = 0

  H = makequeue(V) (using cost-values as keys)
  while H is not empty:
    v = extractMin(H)
    for each z of v's neighbors ({v, z} ∈ E):
      if cost(z) > w(v, z) and H.contains(z):
        cost(z) = w(v, z)
        prev(z) = v
        decreasekey(H, z)
```

- Time complexity? Let's say extractMin and decreaseKey are on binary heaps (O(log(n))
- O(V * log(V) + E * log(V))
  - Because this graph is connected, we know |V| - 1 <= |E| < $|V^2|$, so we can keep the dominant term and get O(E * log(V)).

Efficient implementation

Every vertex has a key and a parent

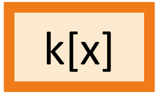
Until all the vertices are **reached:**



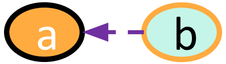
Can't reach x yet

x is “active”

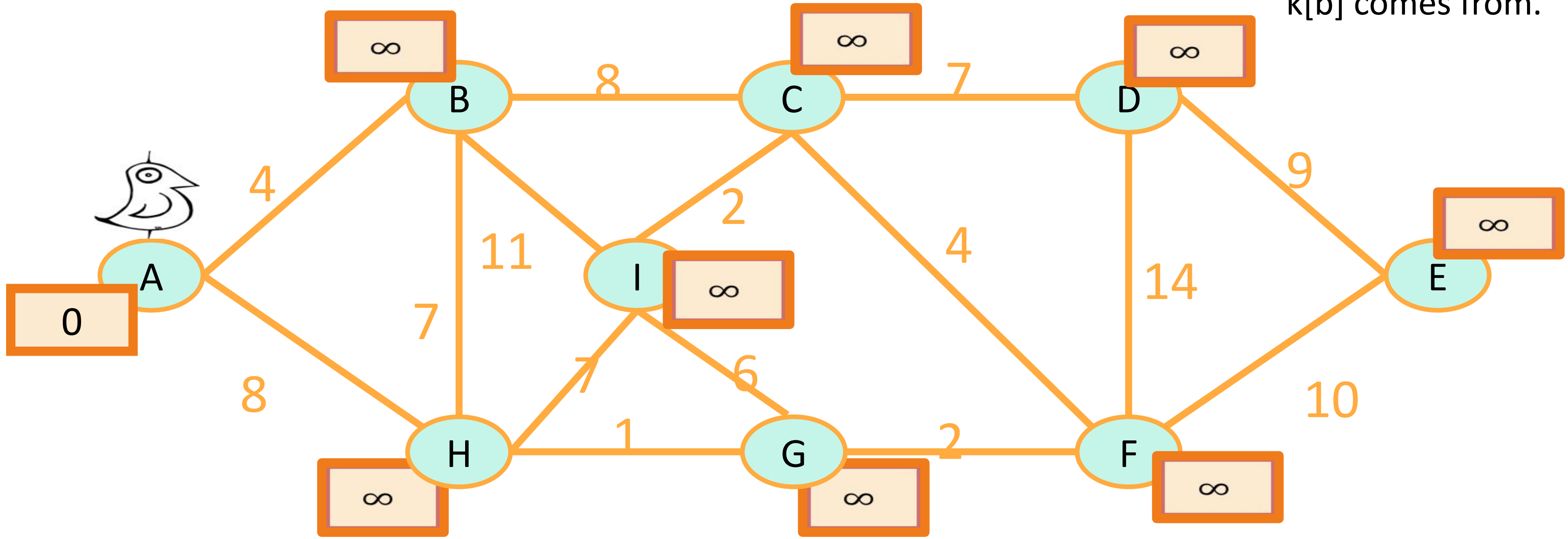
Can reach x



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

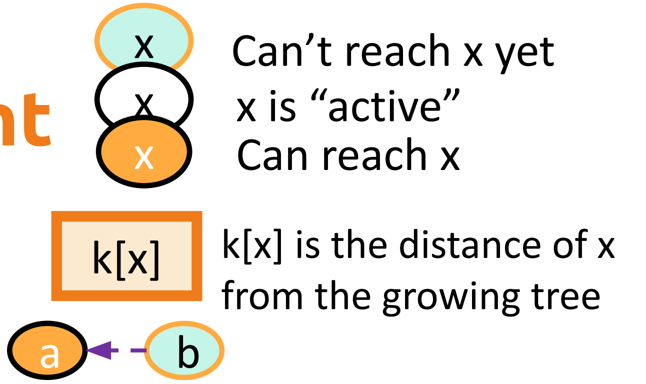


p[b] = a, meaning that a was the vertex that k[b] comes from.



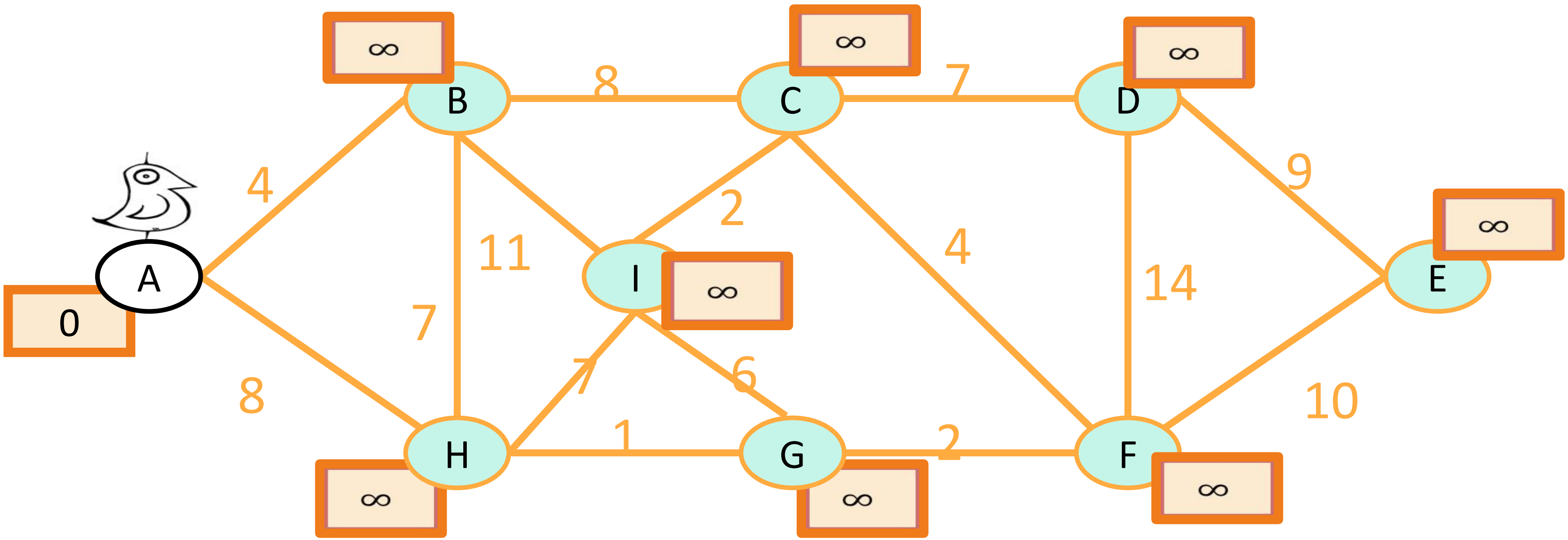
Efficient implementation

Every vertex has a key and a parent



Until all the vertices are **reached:**

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

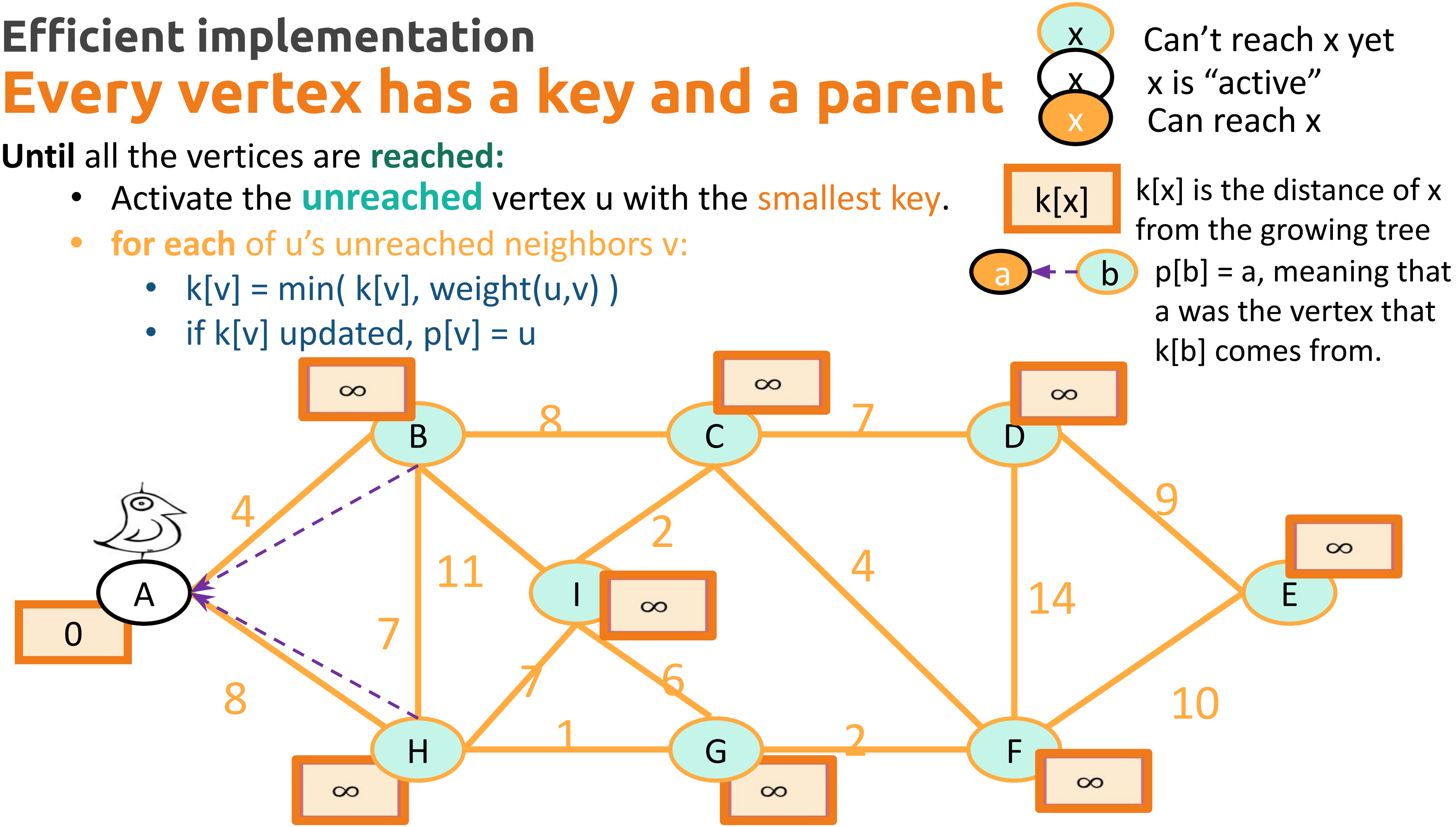


Efficient implementation

Every vertex has a key and a parent

Until all the vertices are **reached:**

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

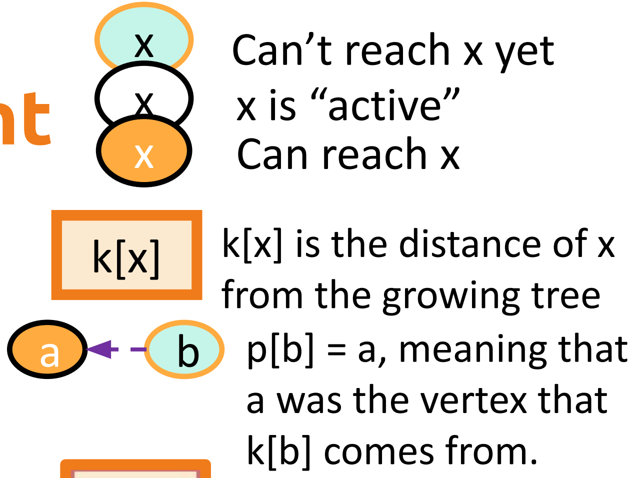


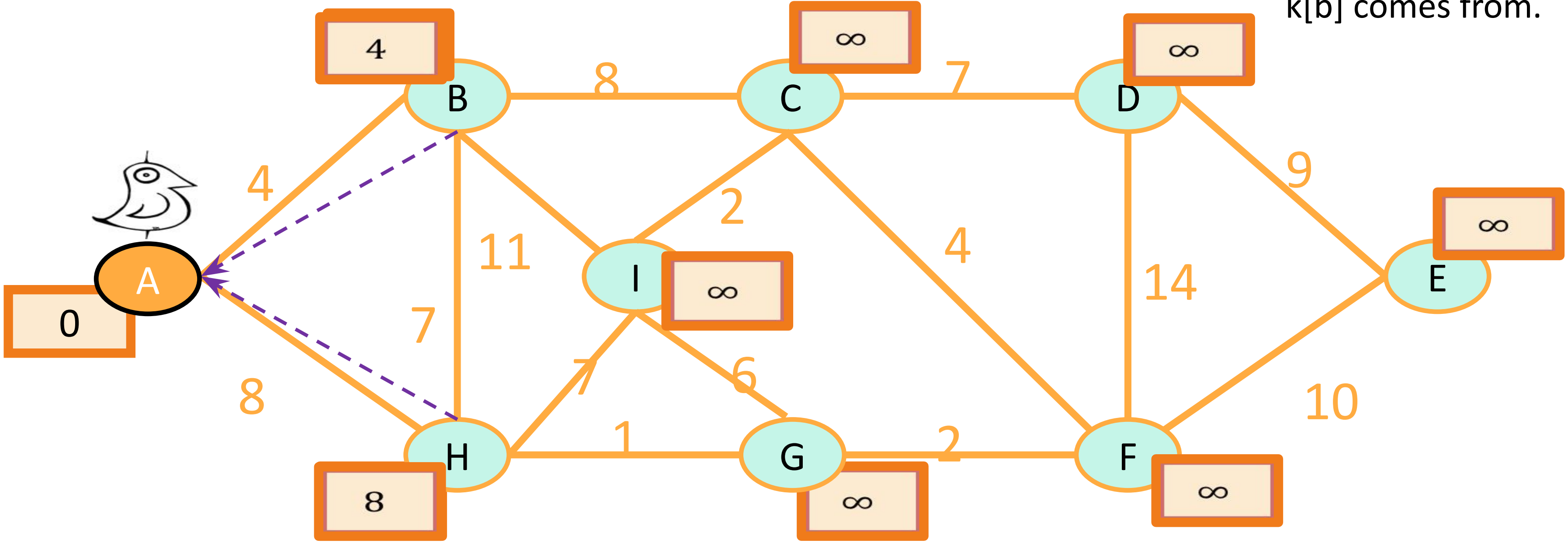
Efficient implementation

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
- Mark u as **reached,** and **add (p[u],u) to MST.**



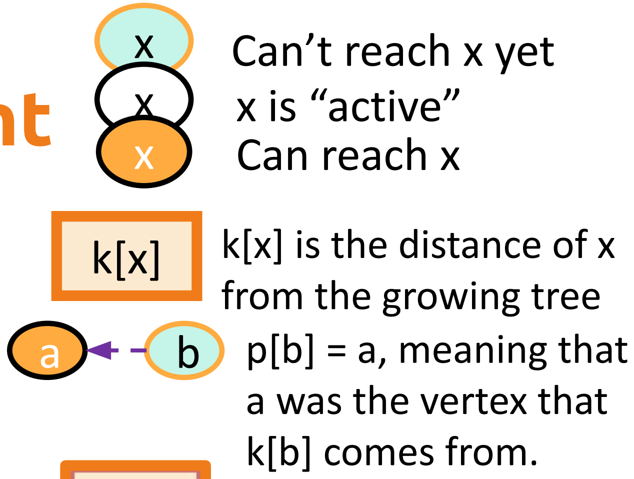


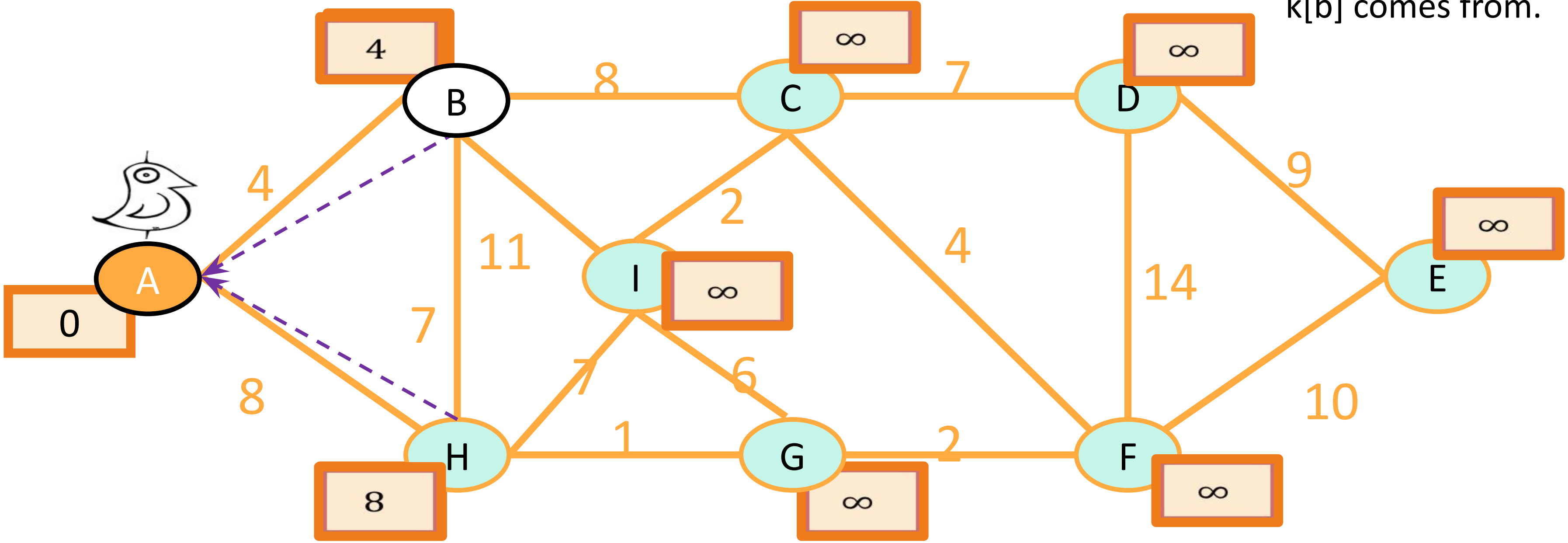
Efficient implementation

Every vertex has a key and a parent

Until all the vertices are **reached:**

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





Efficient implementation

Every vertex has a key and a parent

x Can't reach x yet

x x is "active"

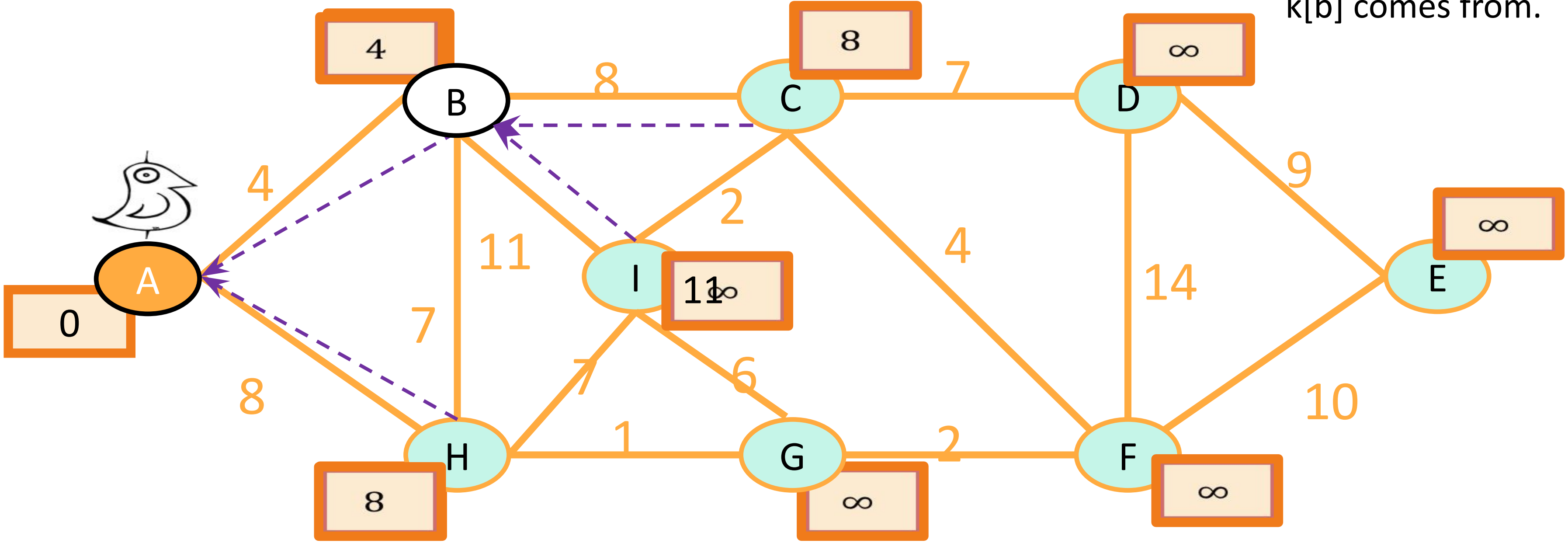
x Can reach x

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

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

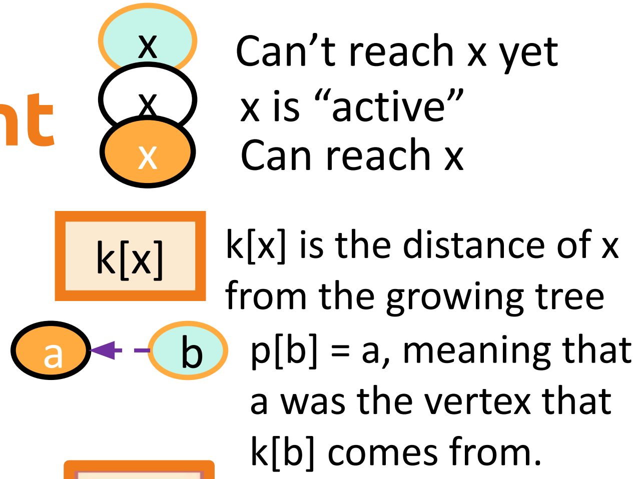
Until all the vertices are **reached:**

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



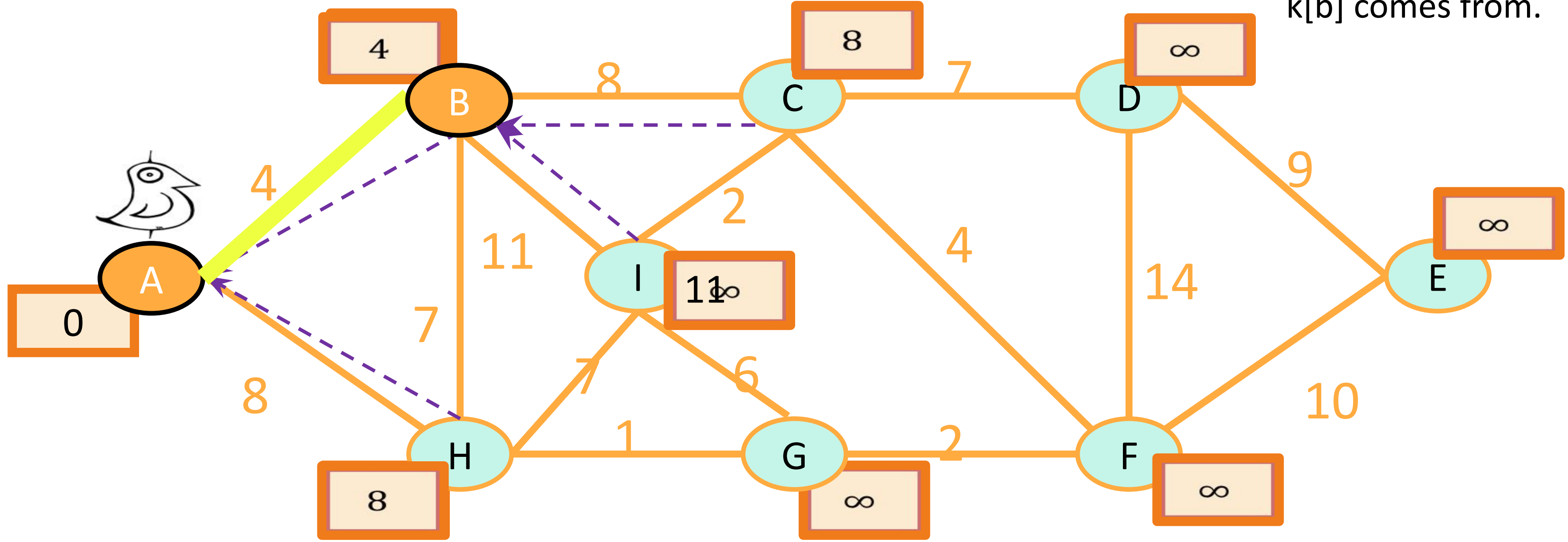
Efficient implementation

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.**



Efficient implementation

Every vertex has a key and a parent

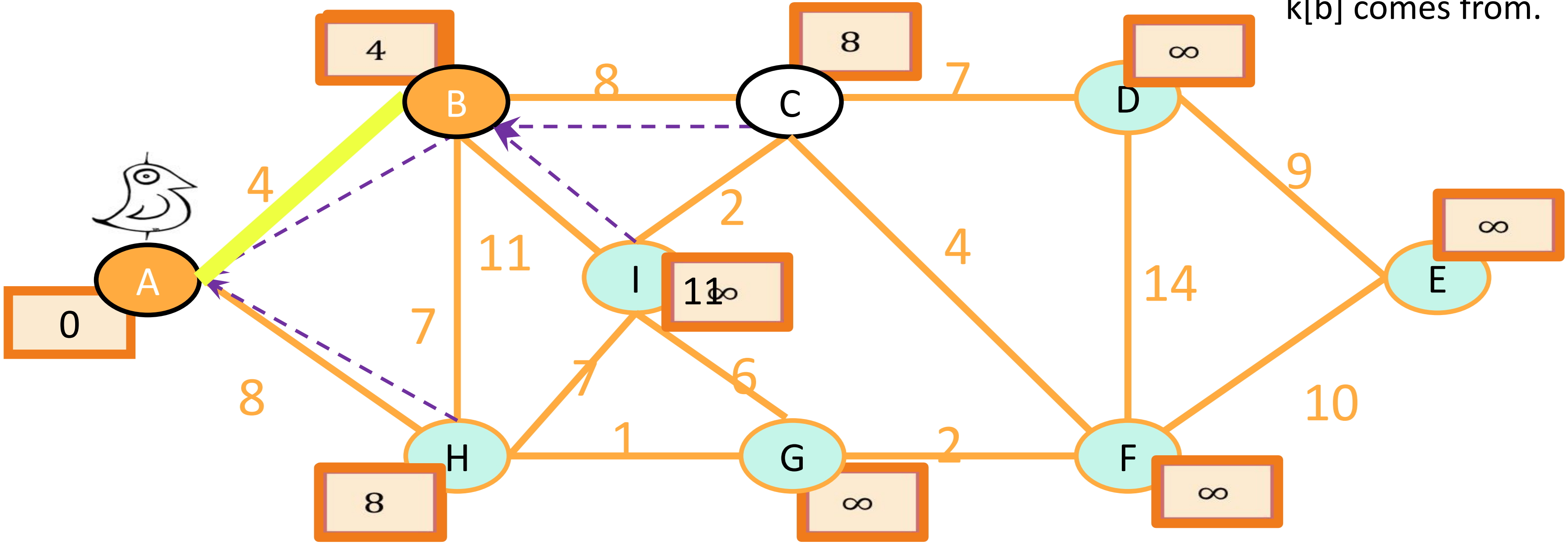
x Can't reach x yet
x x is "active"
x Can reach x

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

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

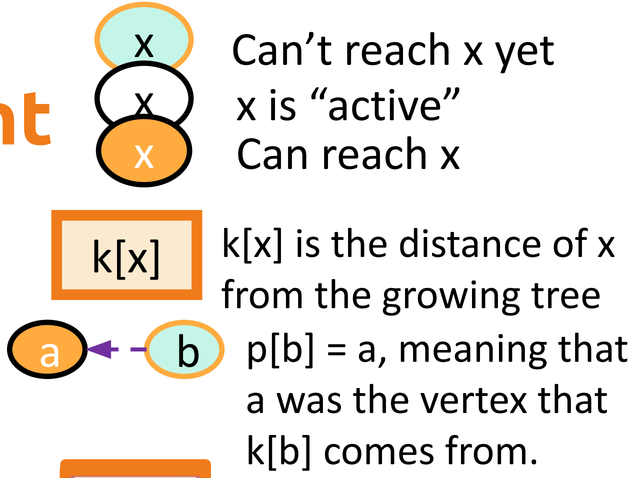
Until all the vertices are **reached:**

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



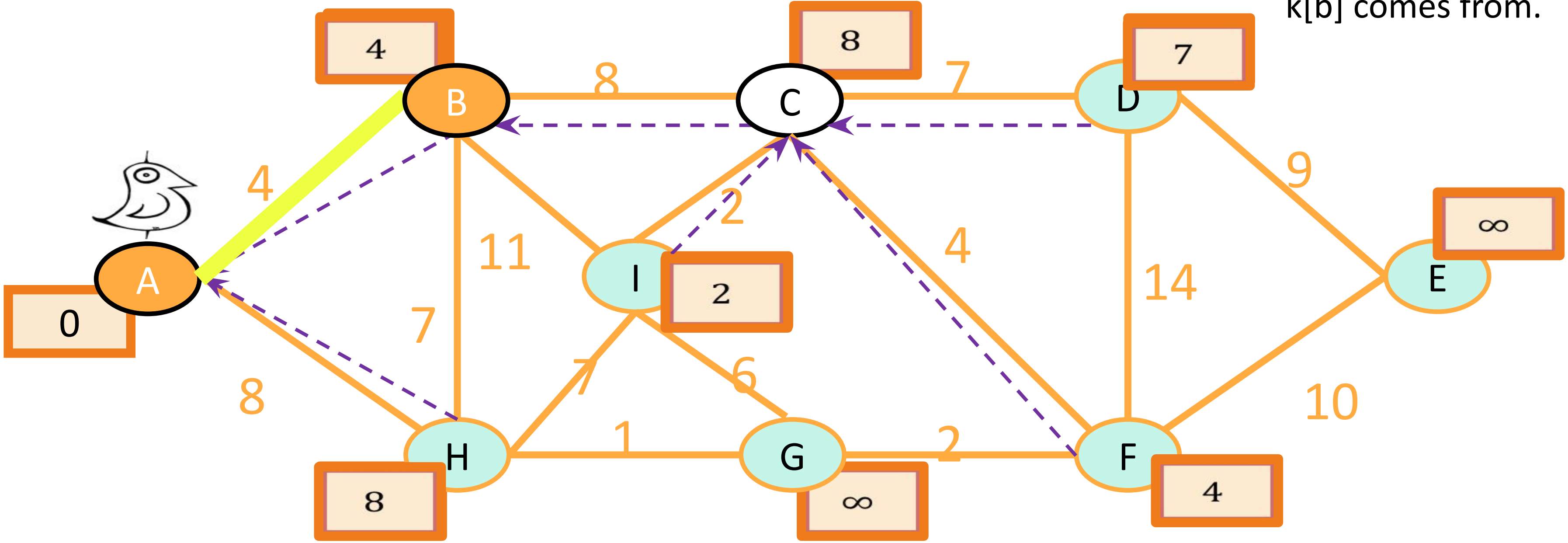
Efficient implementation

Every vertex has a key and a parent



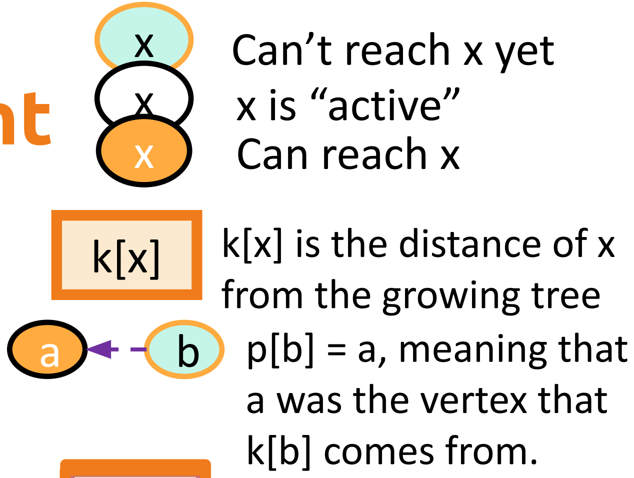
Until all the vertices are **reached:**

- 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.**



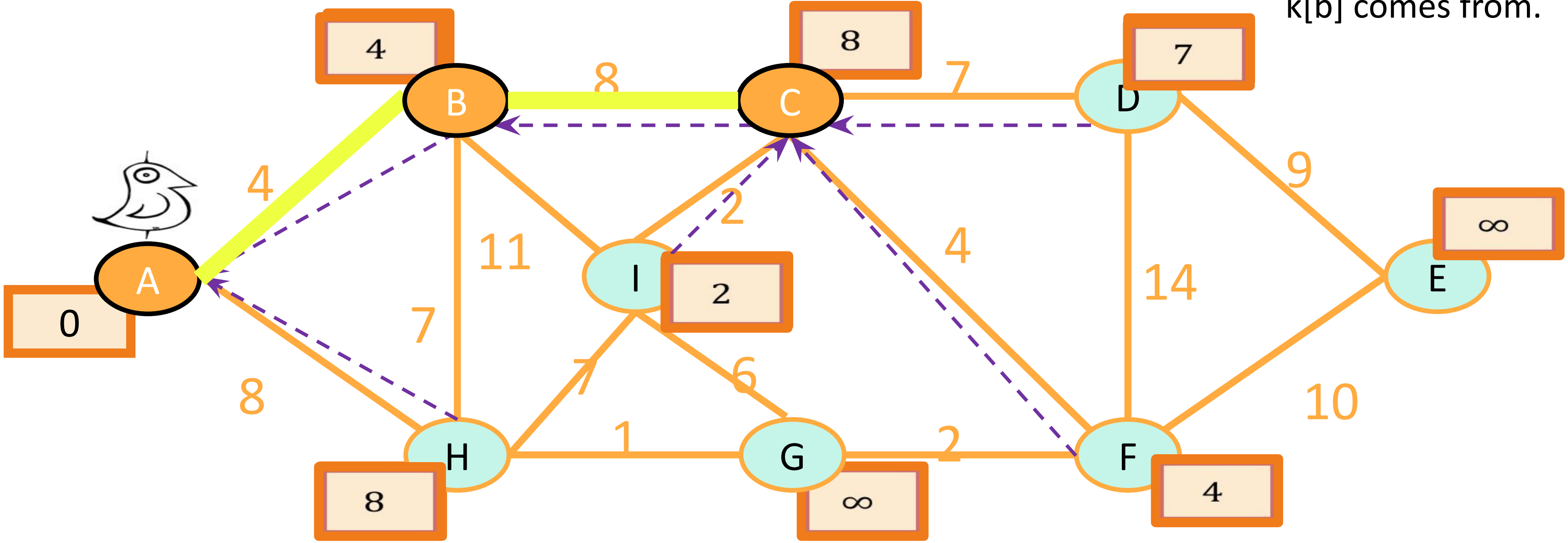
Efficient implementation

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.**



Efficient implementation

Every vertex has a key and a parent

x Can't reach x yet

x x is "active"

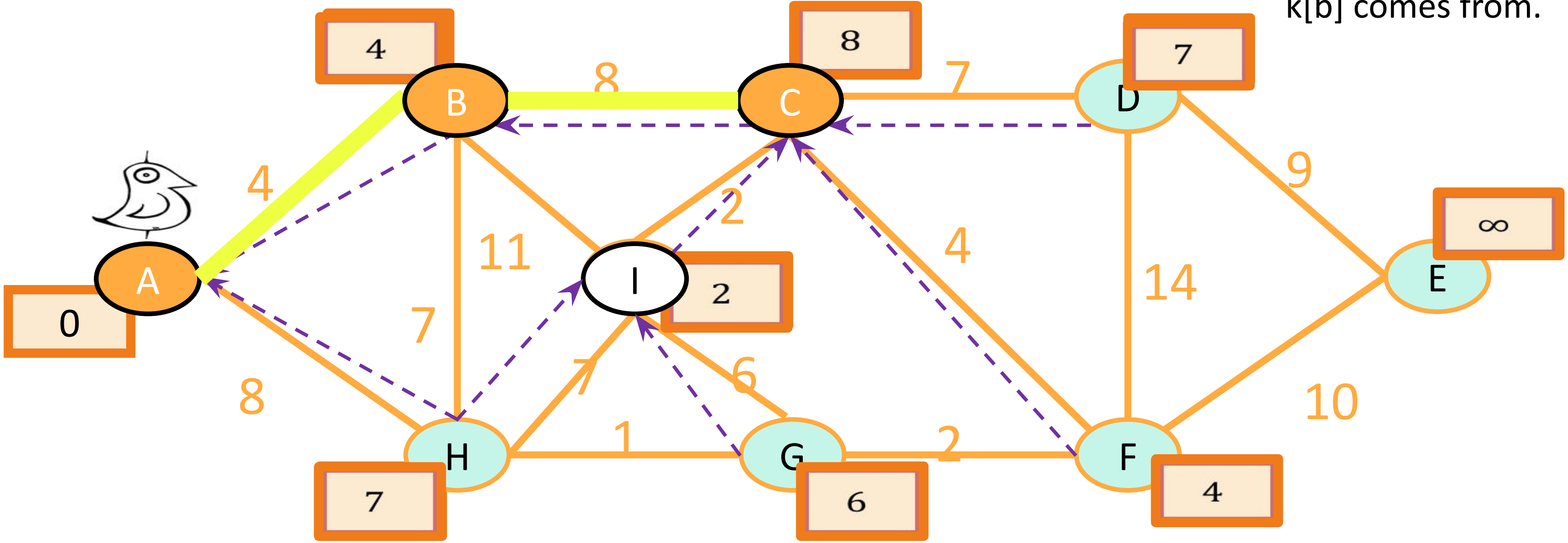
x Can reach x

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

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

Until all the vertices are **reached:**

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



Efficient implementation

Every vertex has a key and a parent

x Can’t reach x yet

x x is “active”

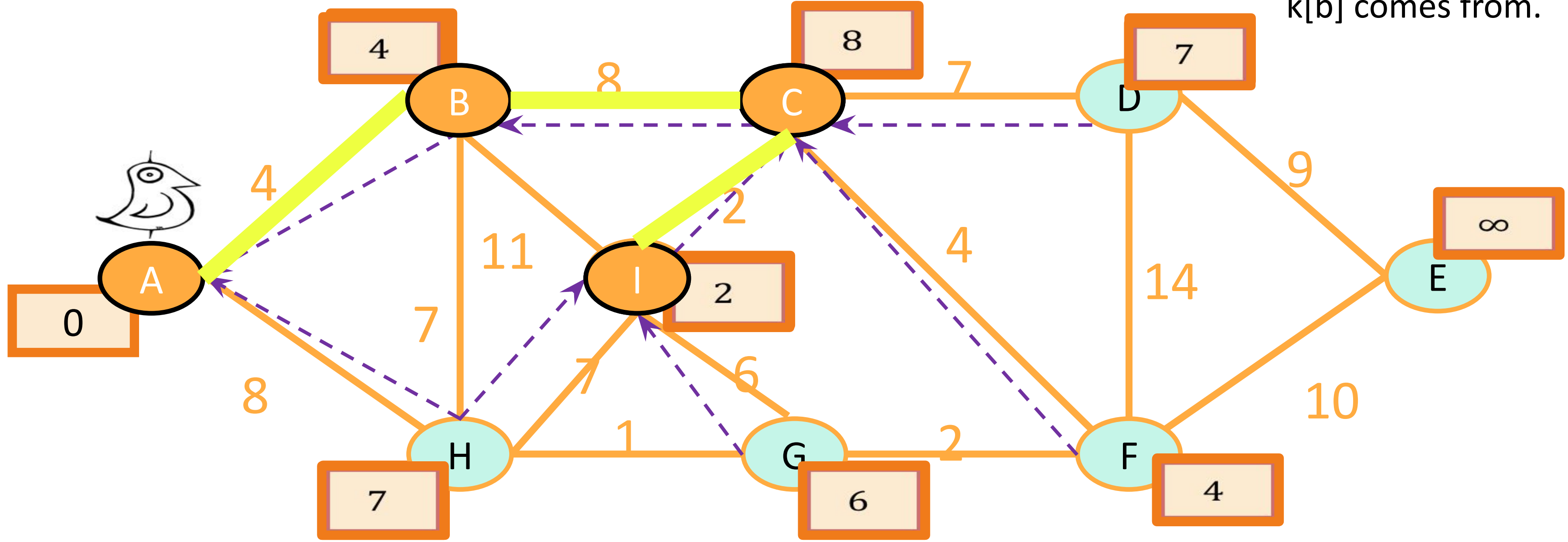
x Can reach x

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

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

Until all the vertices are **reached:**

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

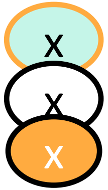


Efficient implementation

Every vertex has a key and a parent

Until all the vertices are **reached:**

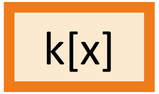
- Activate the **unreached** vertex u with the smallest key.
- **for each** of u's unreached neighbors v:
  - k[v] = min(k[v], weight(u,v))
  - if k[v] updated, p[v] = u
- Mark u as **reached,** and **add (p[u],u) to MST.**



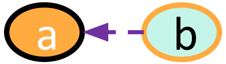
Can't reach x yet

x is "active"

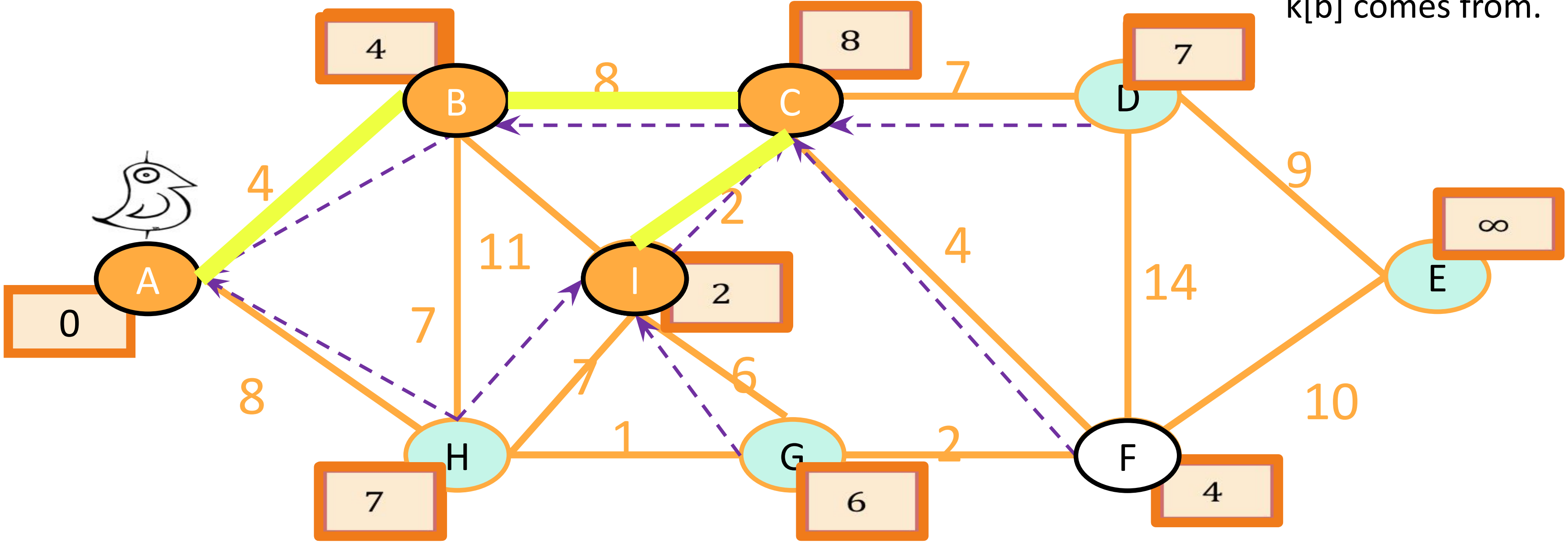
Can reach x



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

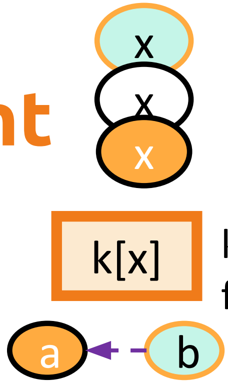


p[b] = a, meaning that a was the vertex that k[b] comes from.



Efficient implementation

Every vertex has a key and a parent



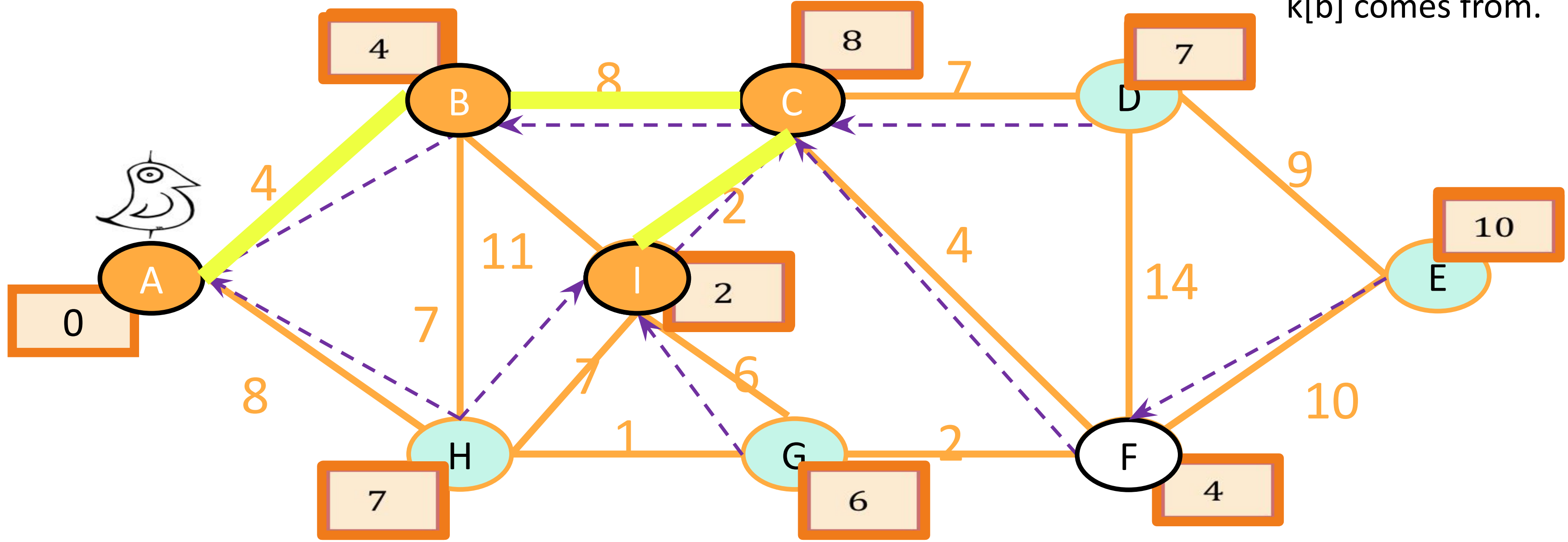
Can't reach x yet
x is "active"
Can reach x

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.

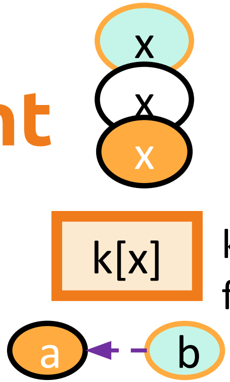
Until all the vertices are **reached:**

- 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
- Mark u as **reached,** and **add (p[u],u) to MST.**



Efficient implementation

Every vertex has a key and a parent



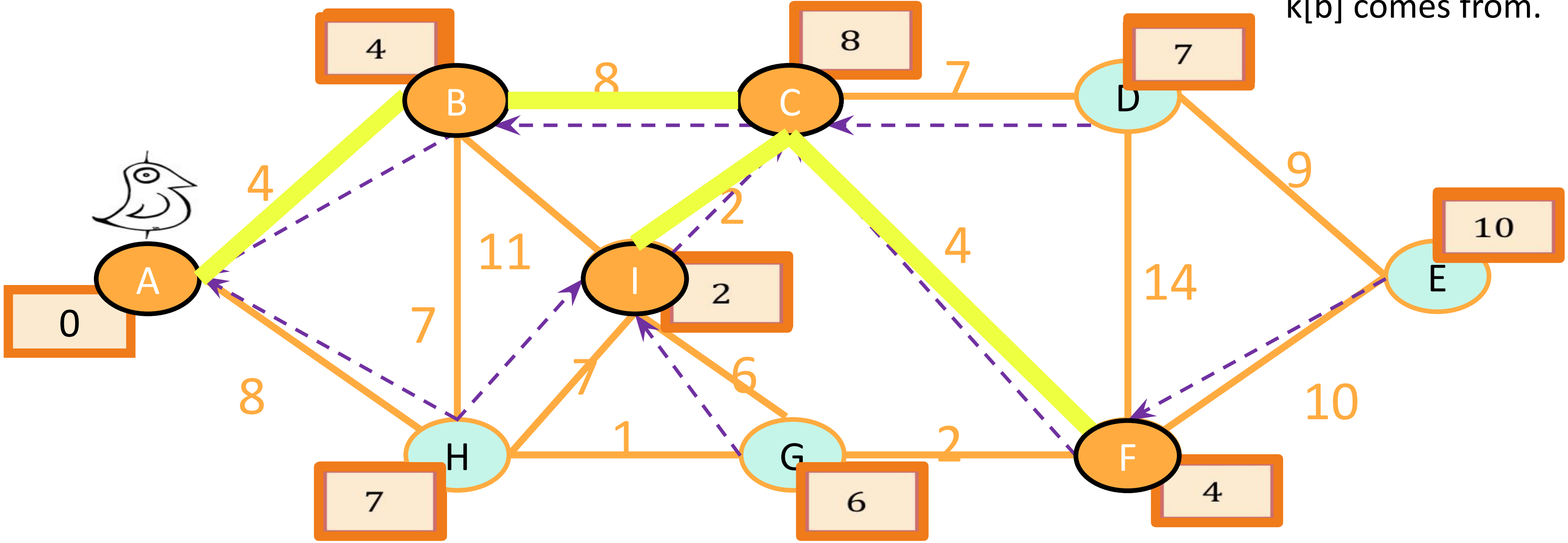
Can’t reach x yet
x is “active”
Can reach x

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
- Mark u as **reached,** and **add (p[u],u) to MST.**

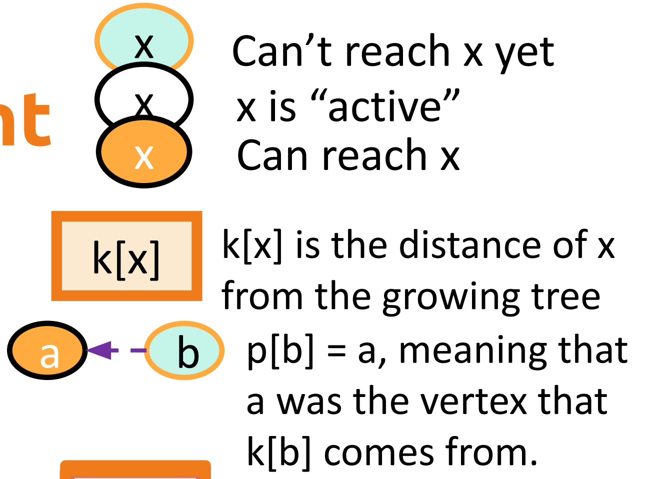


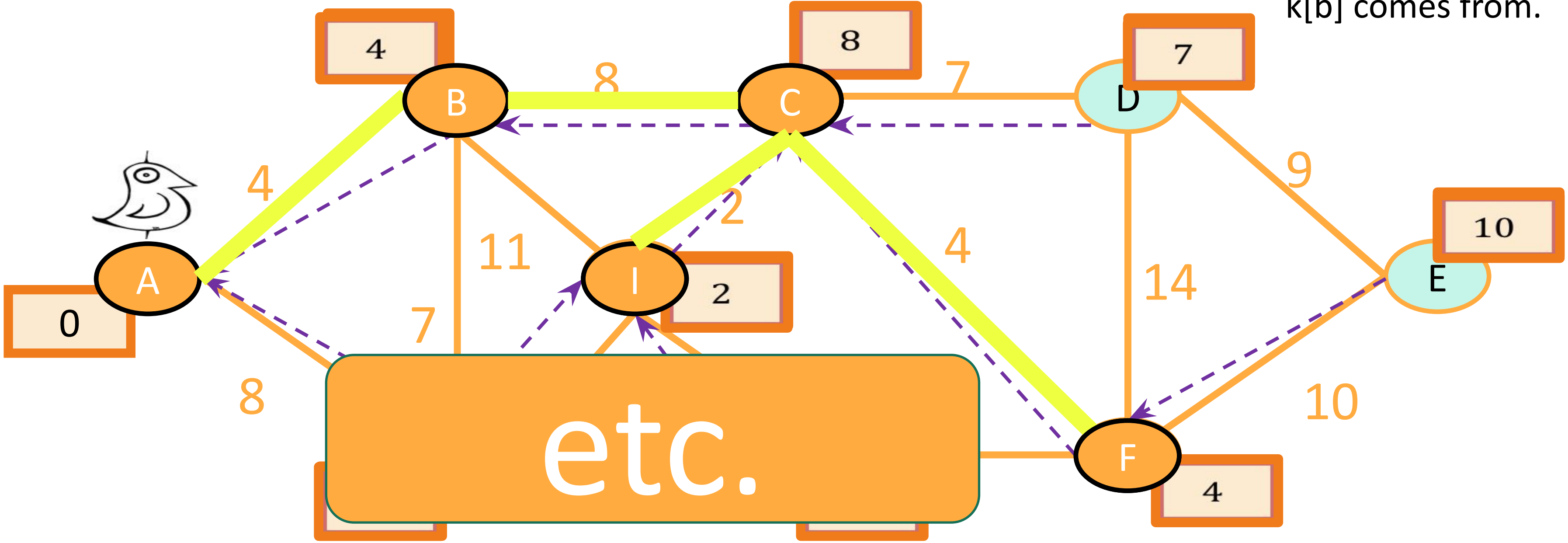
Efficient implementation

Every vertex has a key and a parent

Until all the vertices are **reached:**

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



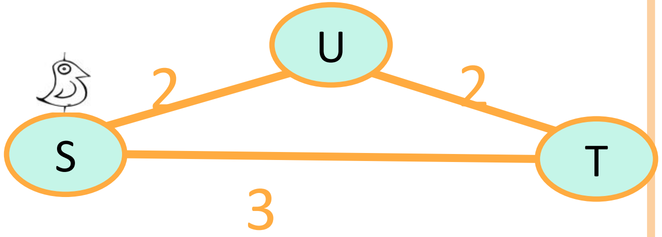


This should look pretty familiar...?

- 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))

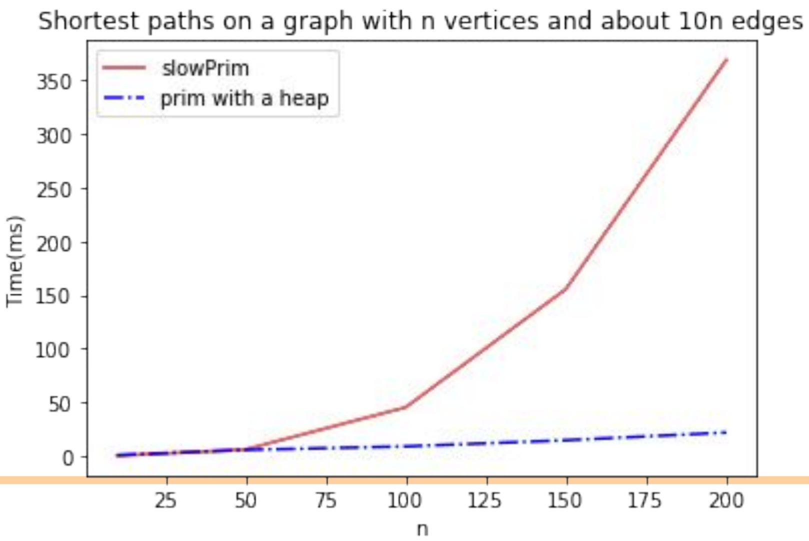
     we keep k[v] which we update by
     - k[v] = min(k[v], w(u,v))
- To see the difference, consider:

Thing 2 is the big difference.



One thing that is similar: Running time

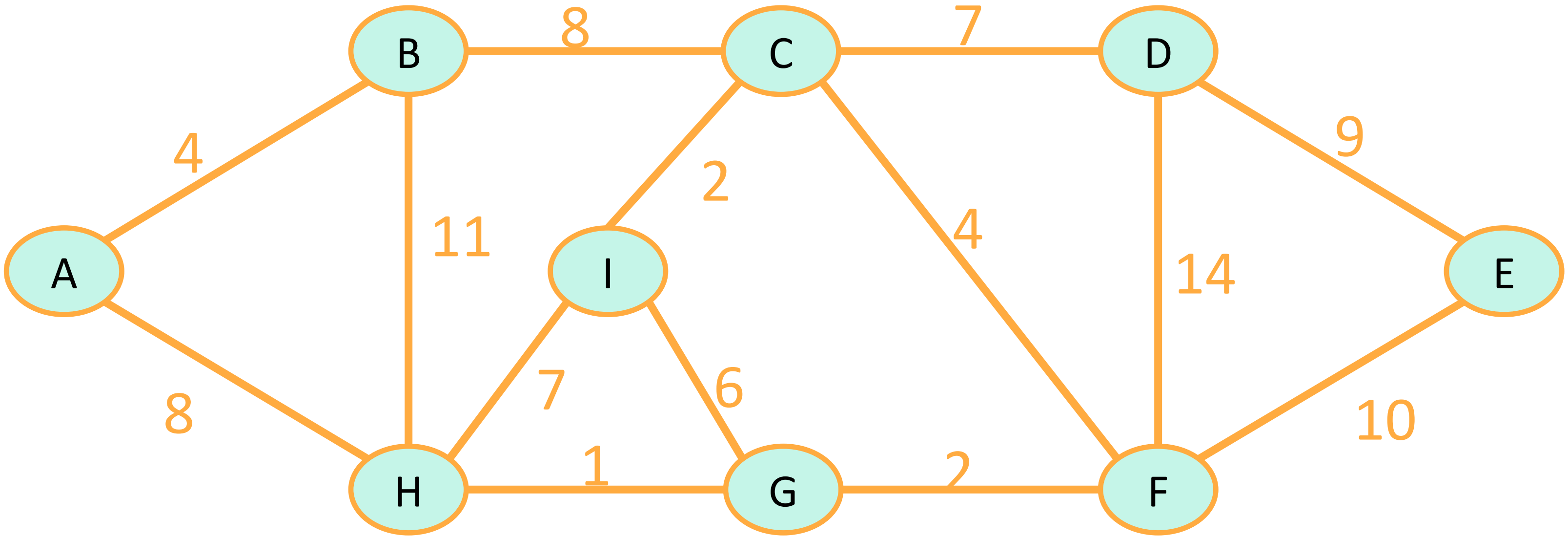
- Exactly the same as Dijkstra:
  - O(mlog(n)) using a balanced tree (as a priority queue).
  - O(m + nlog(n)) amortized time if we use a Fibonacci Heap*.



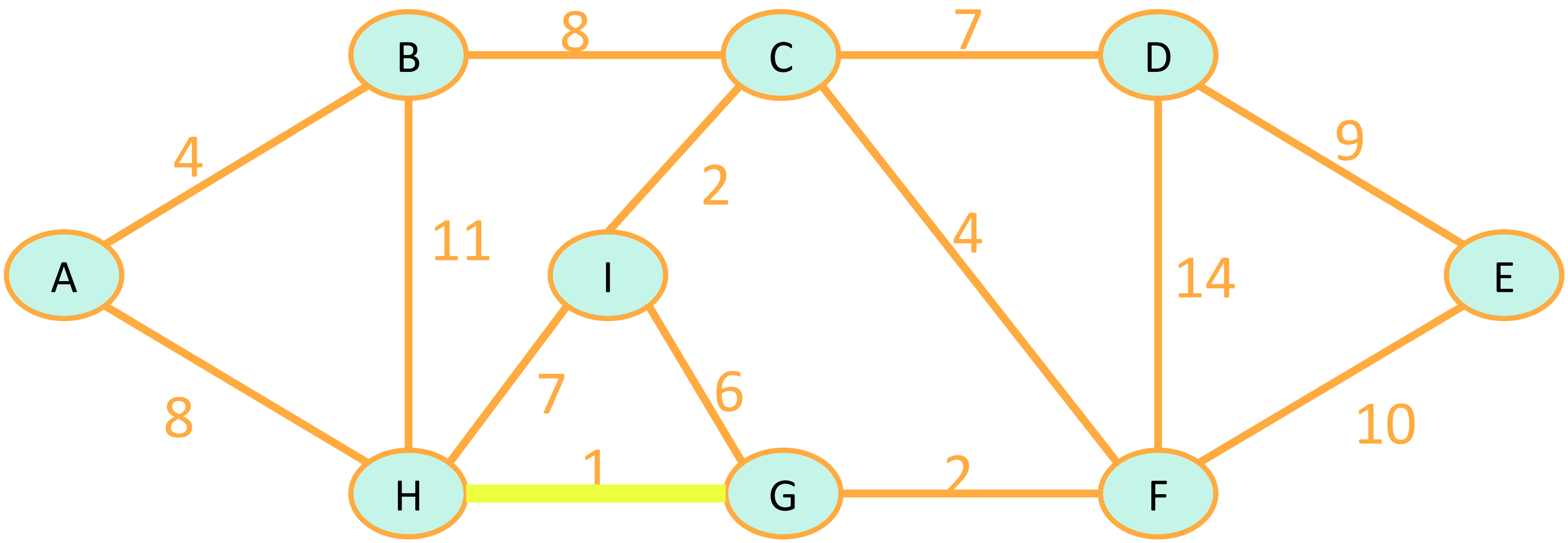
Big Questions!

- What’s a spanning tree/minimum spanning tree?
- How do we find a minimum spanning tree?
- How do we find a minimum spanning tree (pt. 2)?

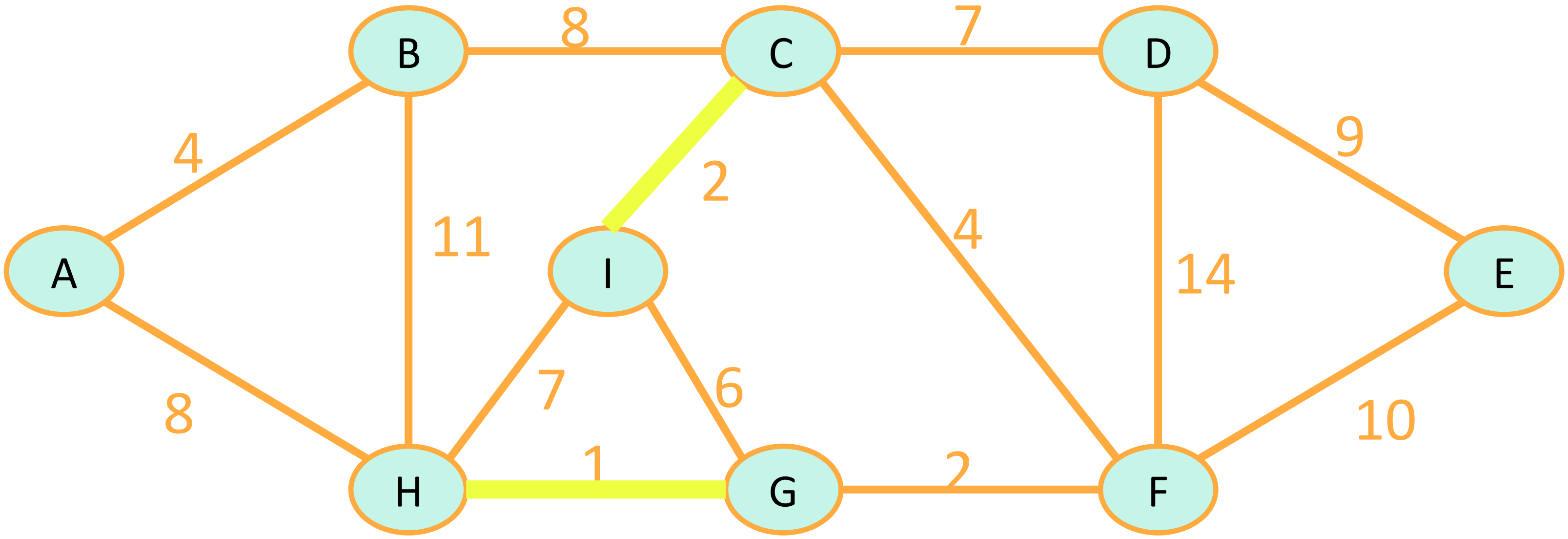
Idea #2: What if we just always take the cheapest edge? Whether or not it's connected to what we have so far?



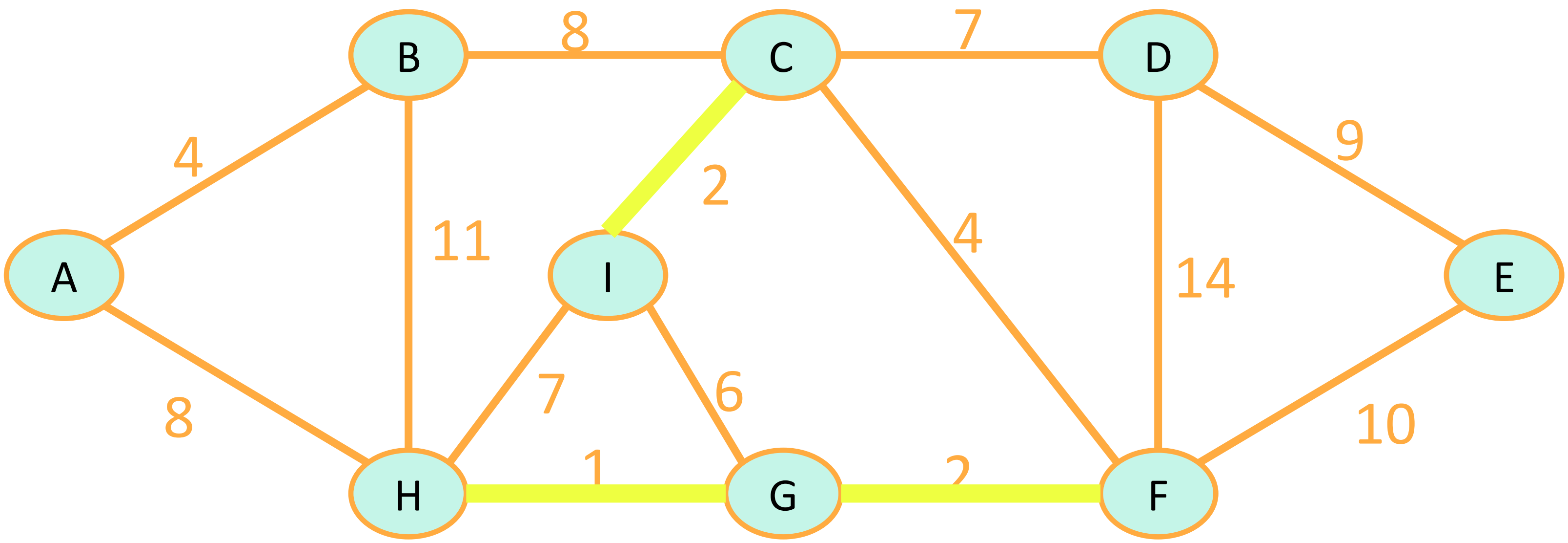
Idea #2: What if we just always take the cheapest edge? Whether or not it's connected to what we have so far?



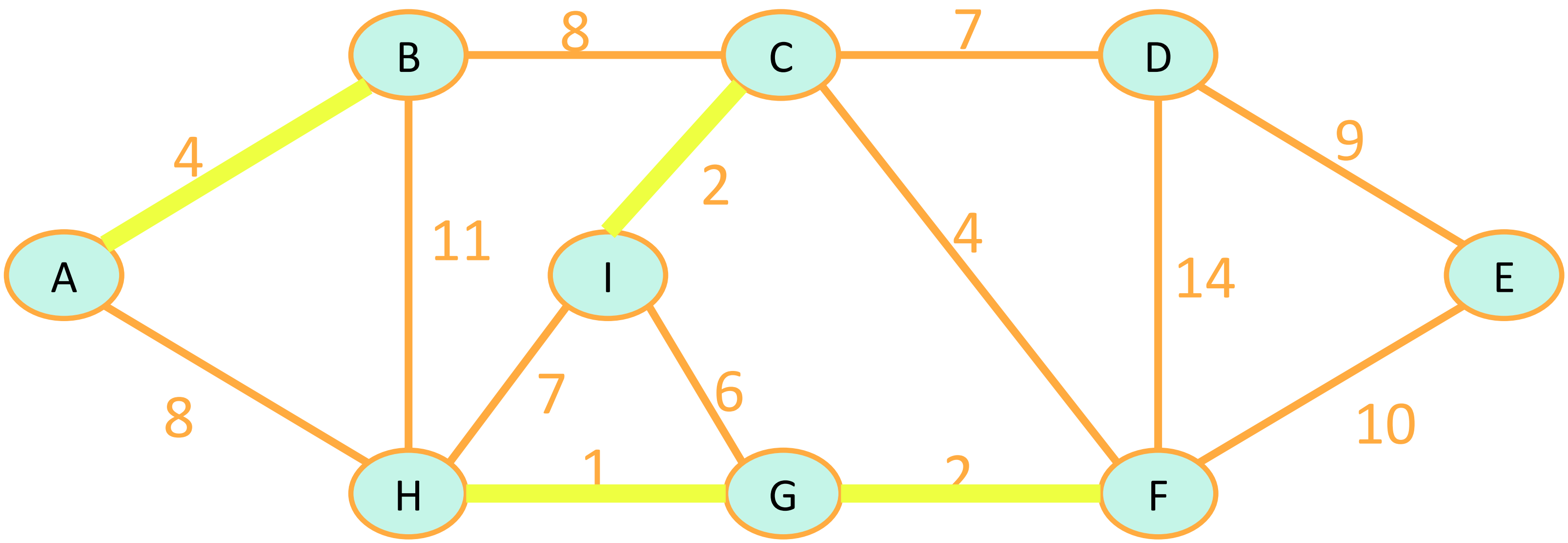
Idea #2: What if we just always take the cheapest edge? Whether or not it's connected to what we have so far?



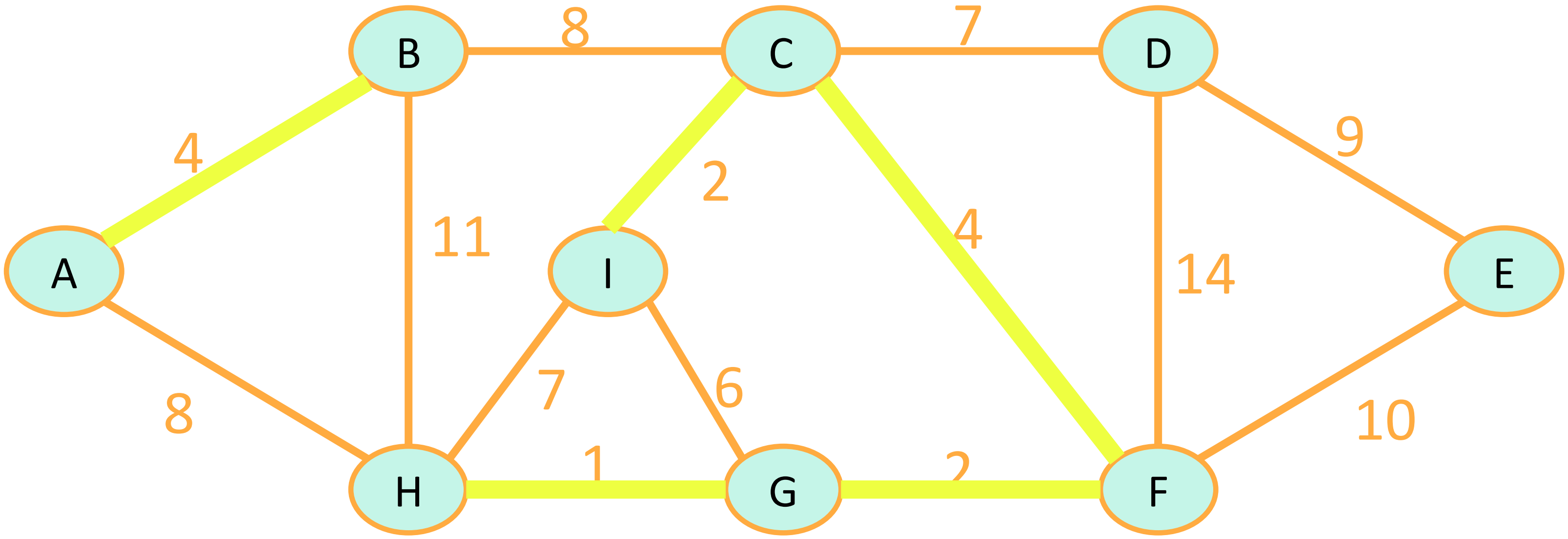
Idea #2: What if we just always take the cheapest edge?
Whether or not it's connected to what we have so far?



Idea #2: What if we just always take the cheapest edge? Whether or not it's connected to what we have so far?

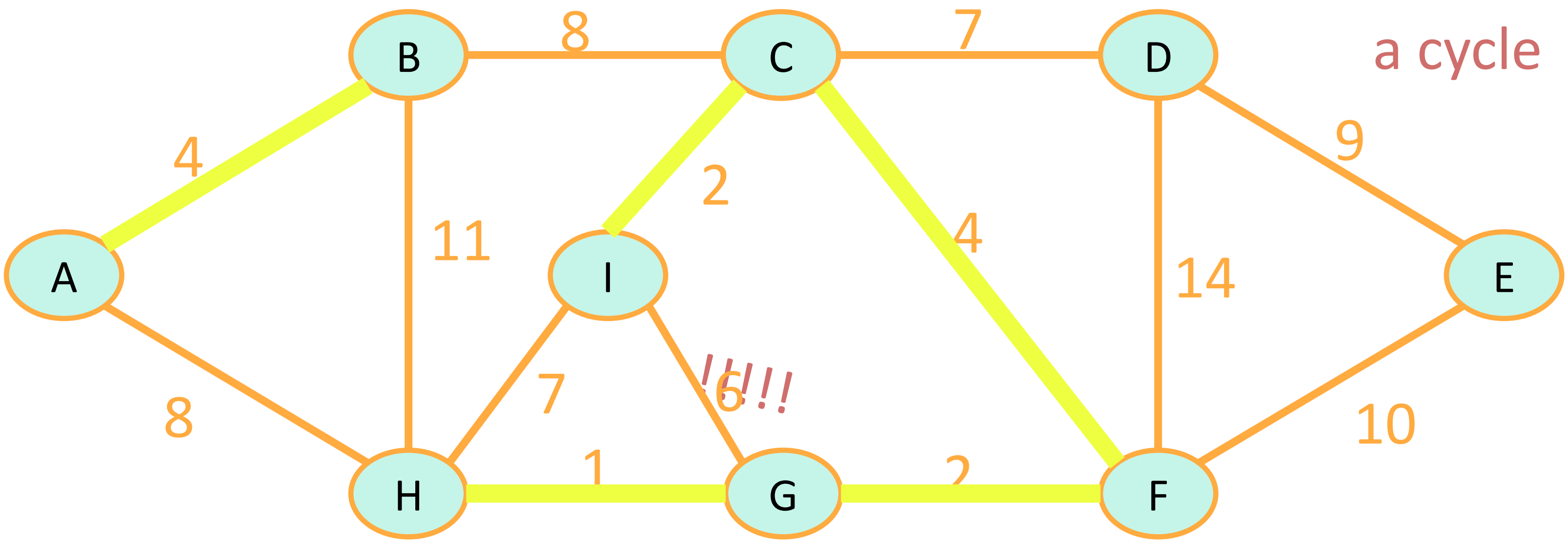


Idea #2: What if we just always take the cheapest edge? Whether or not it's connected to what we have so far?



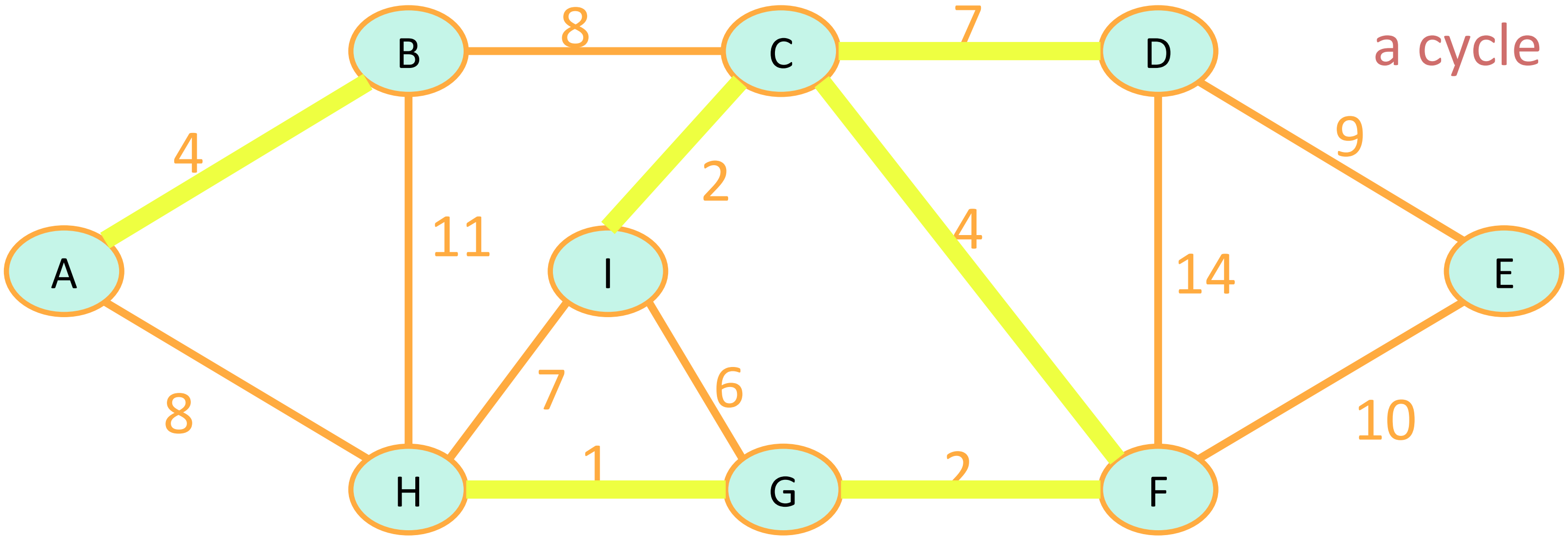
Idea #2: What if we just always take the cheapest edge?
Whether or not it's connected to what we have so far?

That won't cau
a cycle



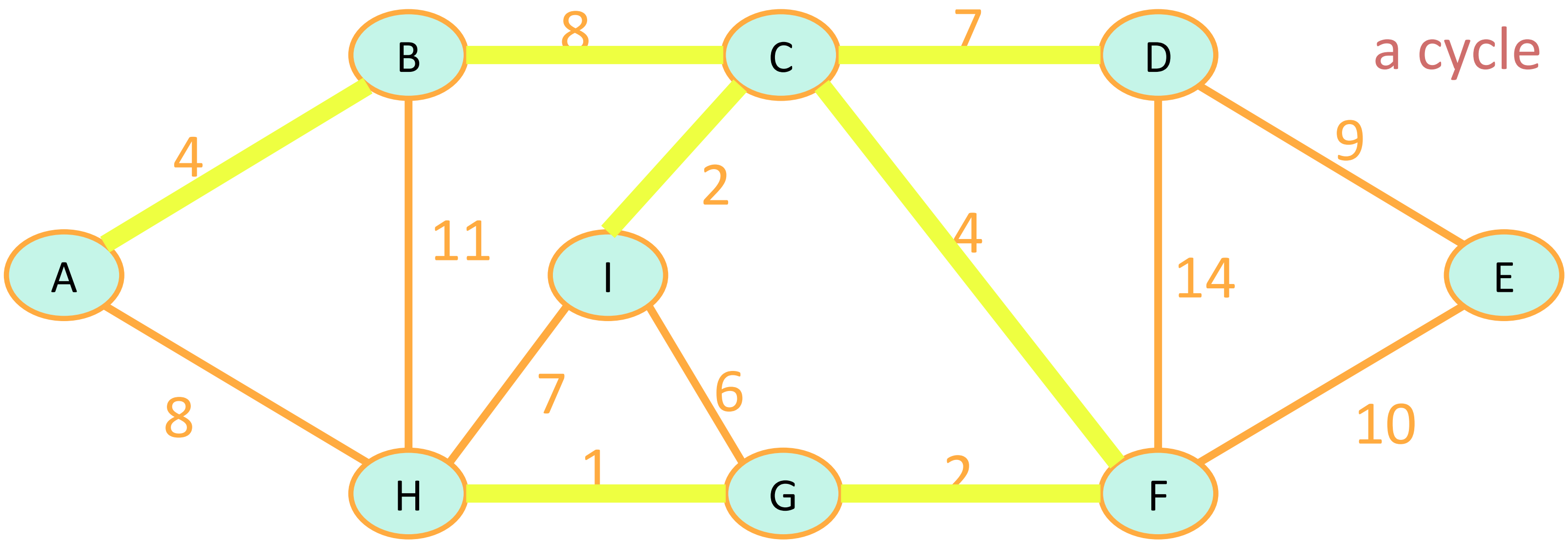
Idea #2: What if we just always take the cheapest edge? Whether or not it's connected to what we have so far?

That won't cau
a cycle



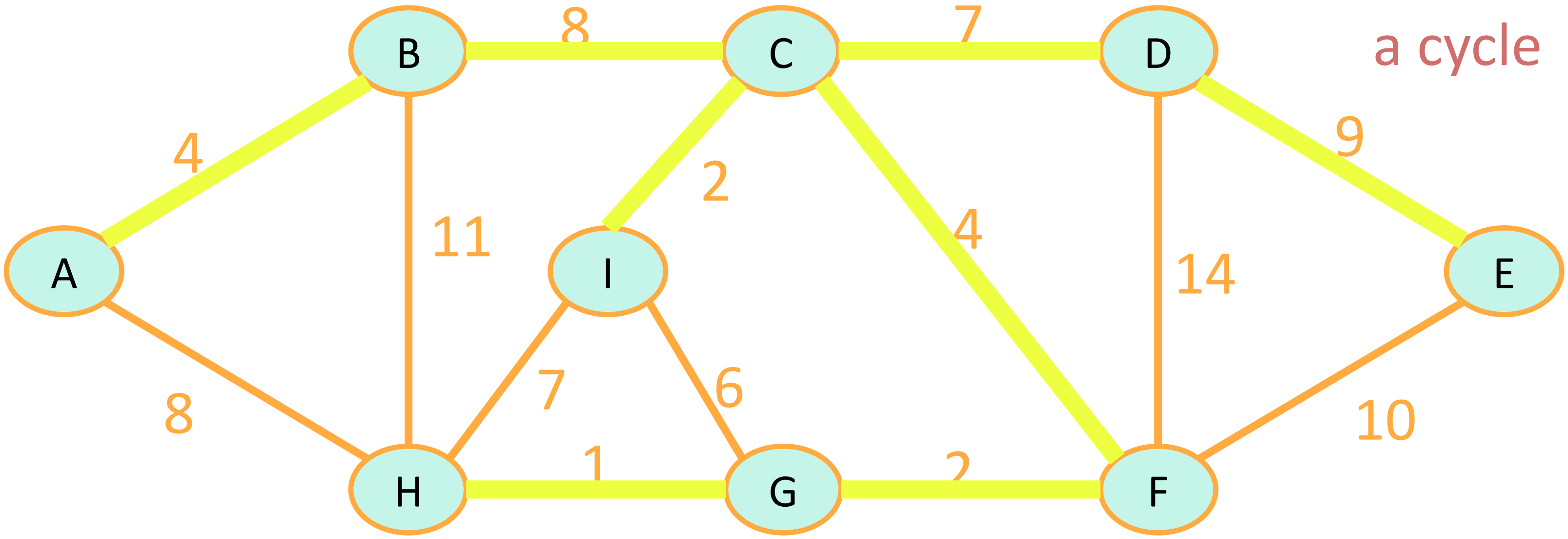
Idea #2: What if we just always take the cheapest edge? Whether or not it's connected to what we have so far?

That won't cau
a cycle



Idea #2: What if we just always take the cheapest edge?
Whether or not it's connected to what we have so far?

That won't ca[u]
a cycle



We’ve Discovered Kruskal’s Algorithm!

- KruskalLite(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?

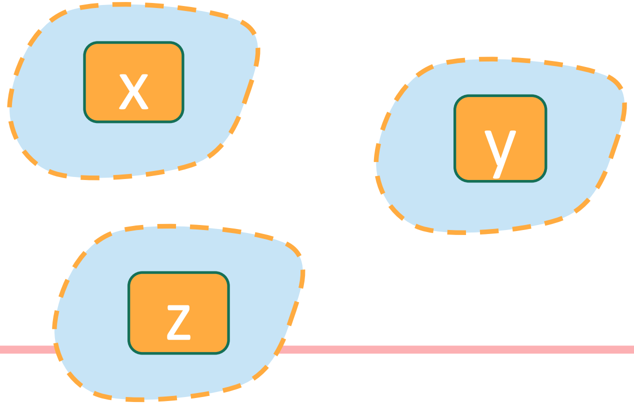
Keep the trees in a special data structure...

"treehouse"?

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

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

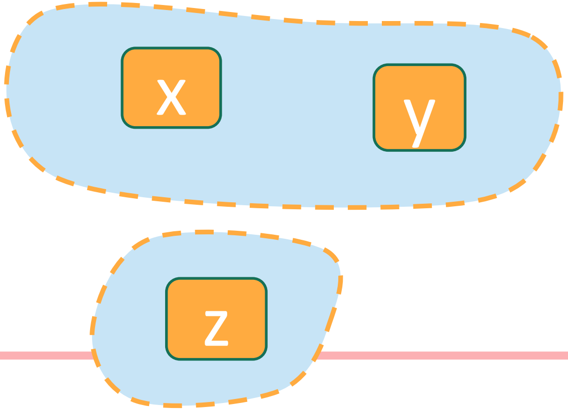
```
makeSet(x)
makeSet(y)
makeSet(z)
union(x,y)
```



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

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

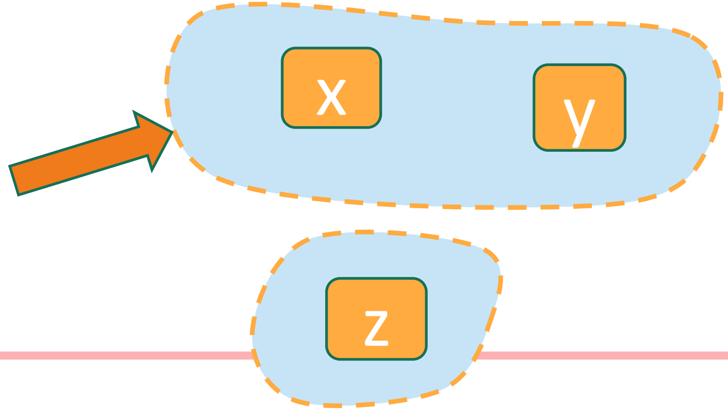
```
makeSet(x)
makeSet(y)
makeSet(z)
union(x,y)
```



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

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

```
makeSet(x)
makeSet(y)
makeSet(z)
union(x,y)
find(x)
```



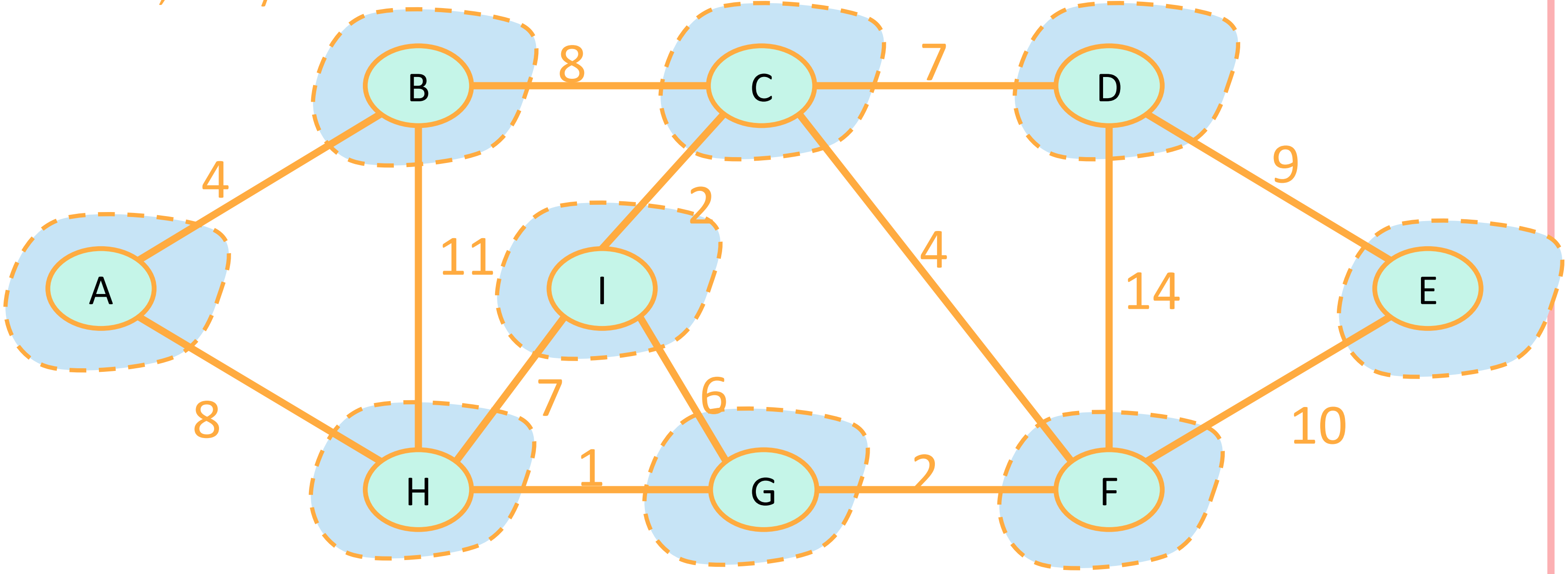
Kruskal Pseudocode (Take 2)

- **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

At each step of Kruskal's, we maintain a forest (collection of trees).

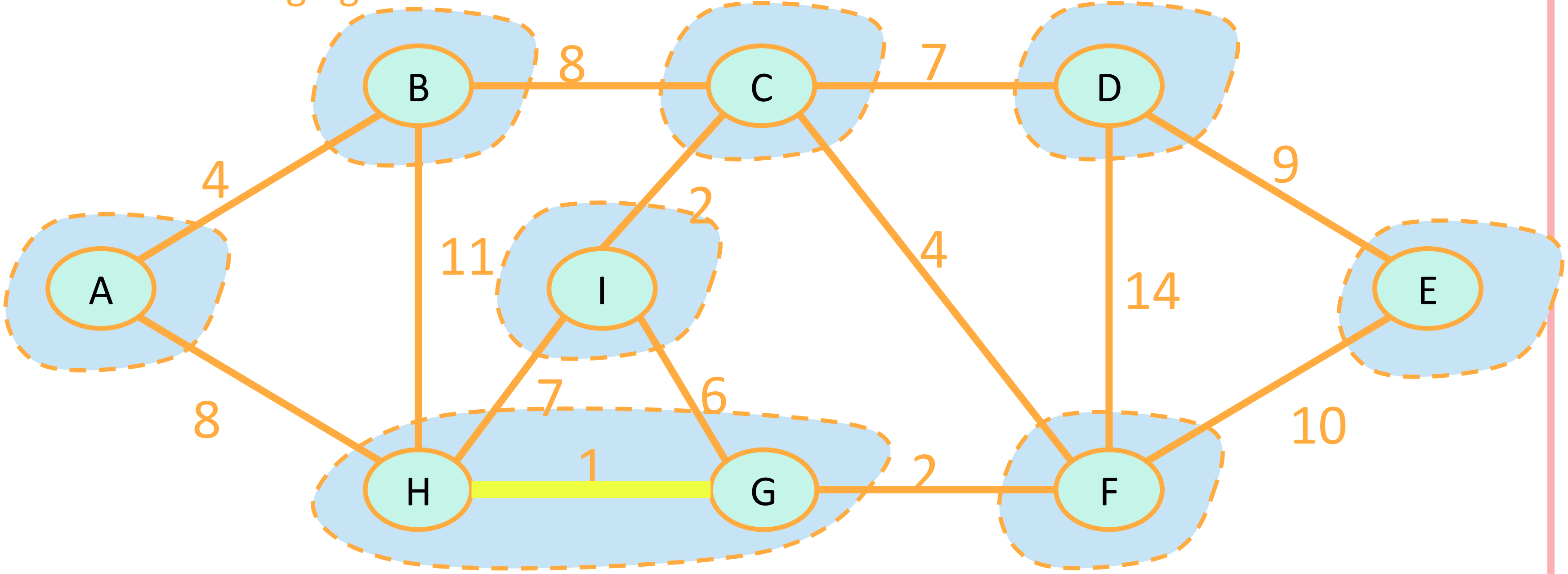
At each step of Kruskal's, we maintain a forest (collection of trees).

To start, every vertex is in its own tree.



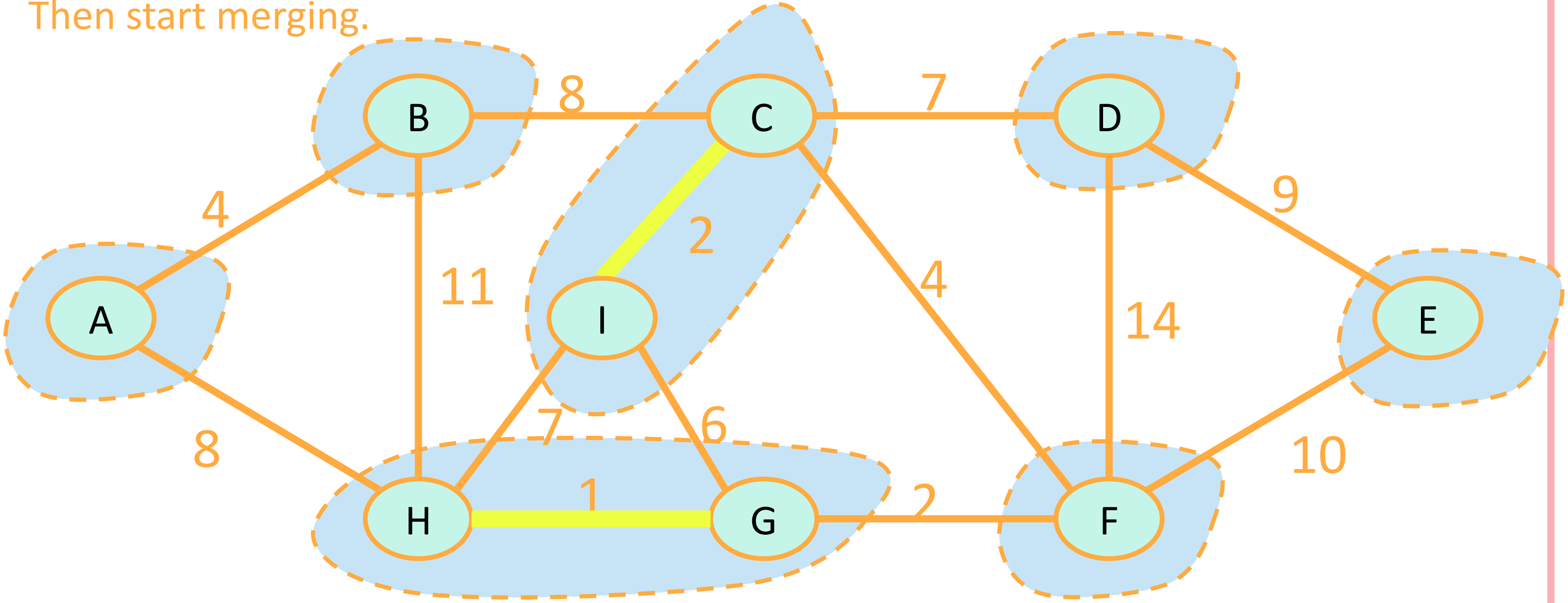
At each step of Kruskal's, we maintain a forest (collection of trees).

Then start merging.



At each step of Kruskal's, we maintain a forest (collection of trees).

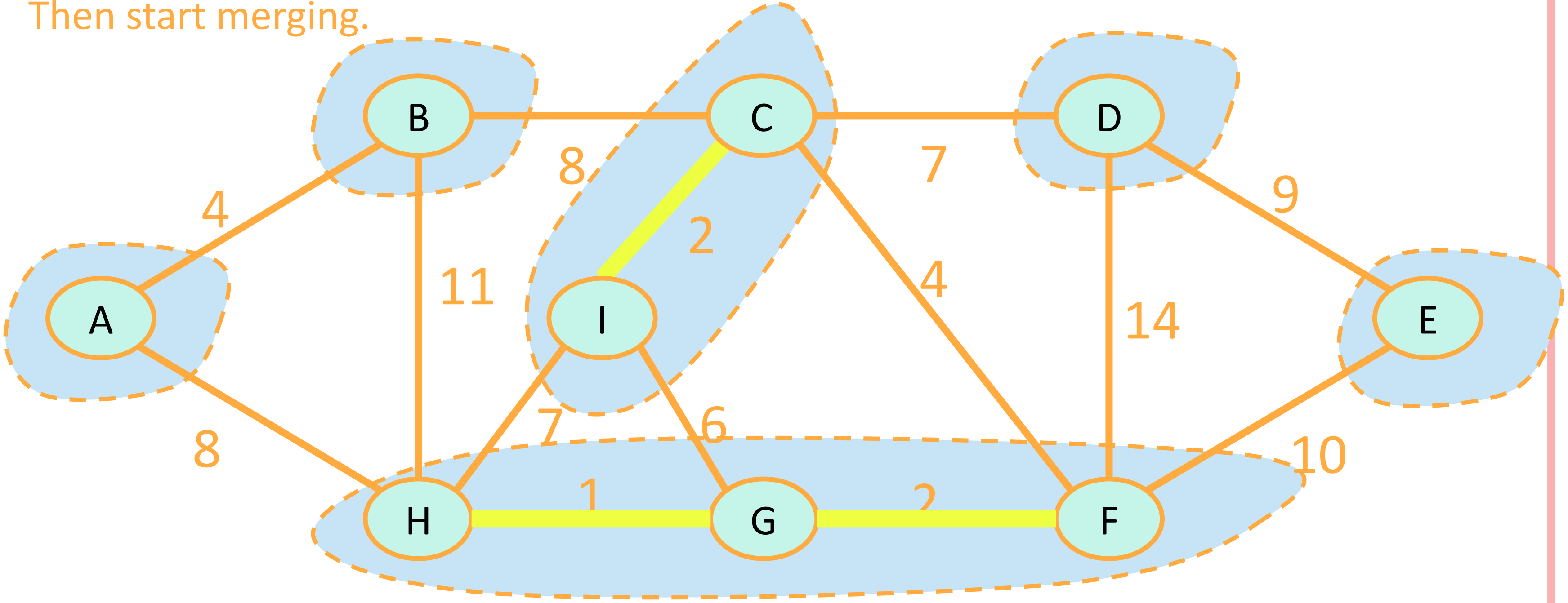
Then start merging.



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

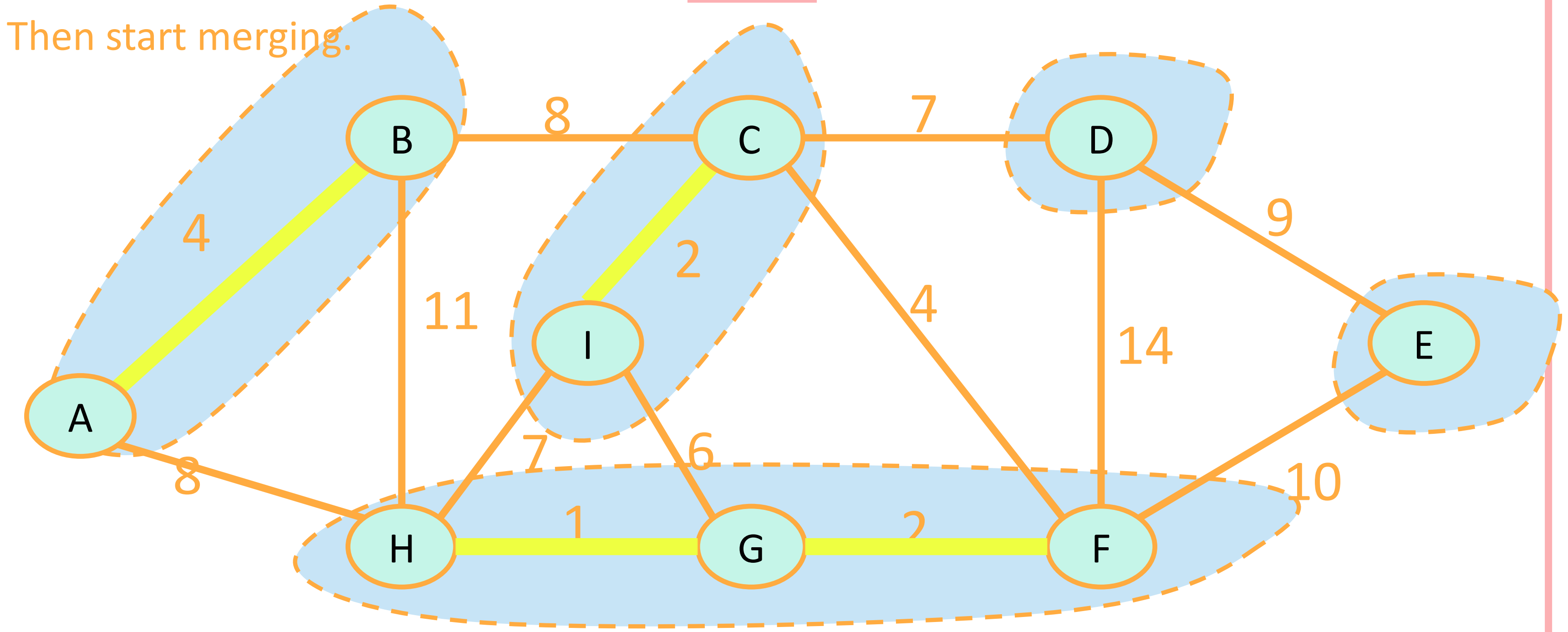
At each step of Kruskal's, we maintain a forest (collection of trees).

Then start merging.



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

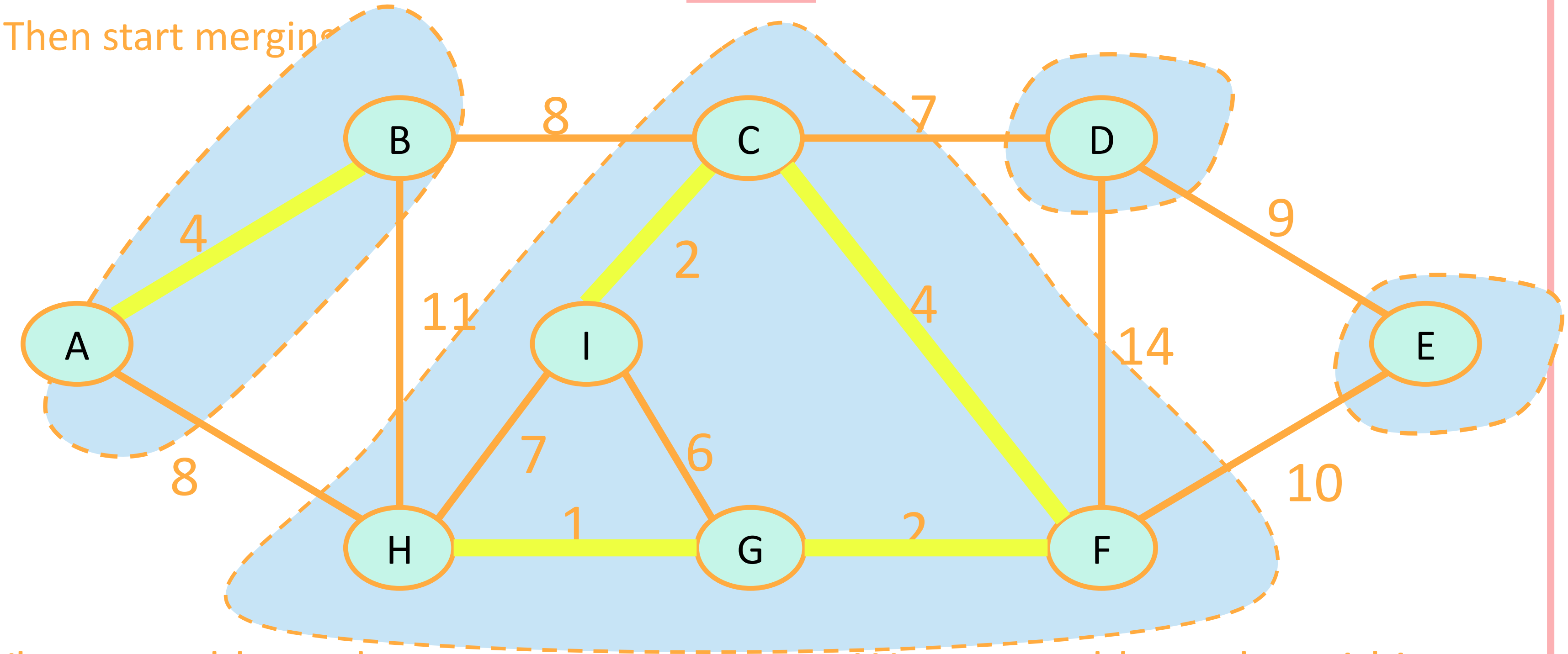
At each step of Kruskal's, we maintain a forest (collection of trees).



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

At each step of Kruskal's, we maintain a forest (collection of trees).

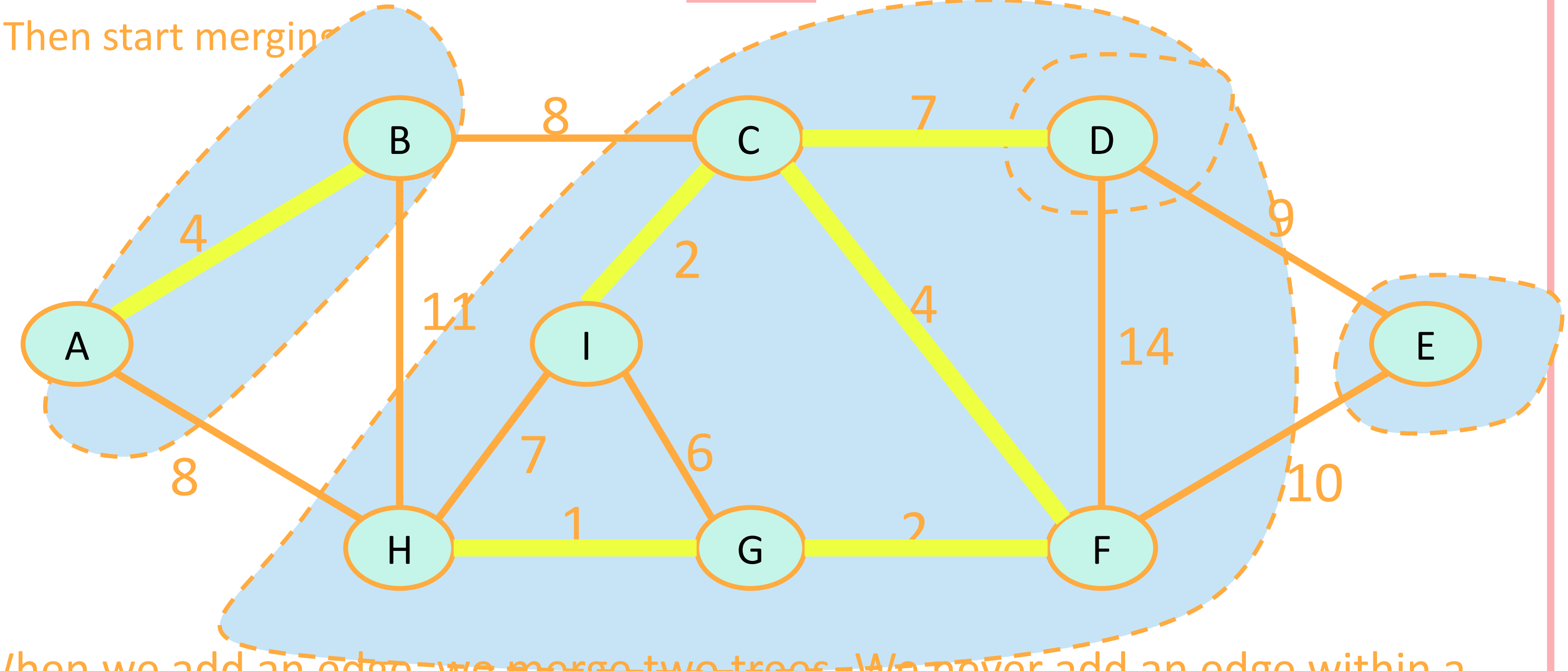
Then start merging



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

At each step of Kruskal's, we maintain a forest (collection of trees).

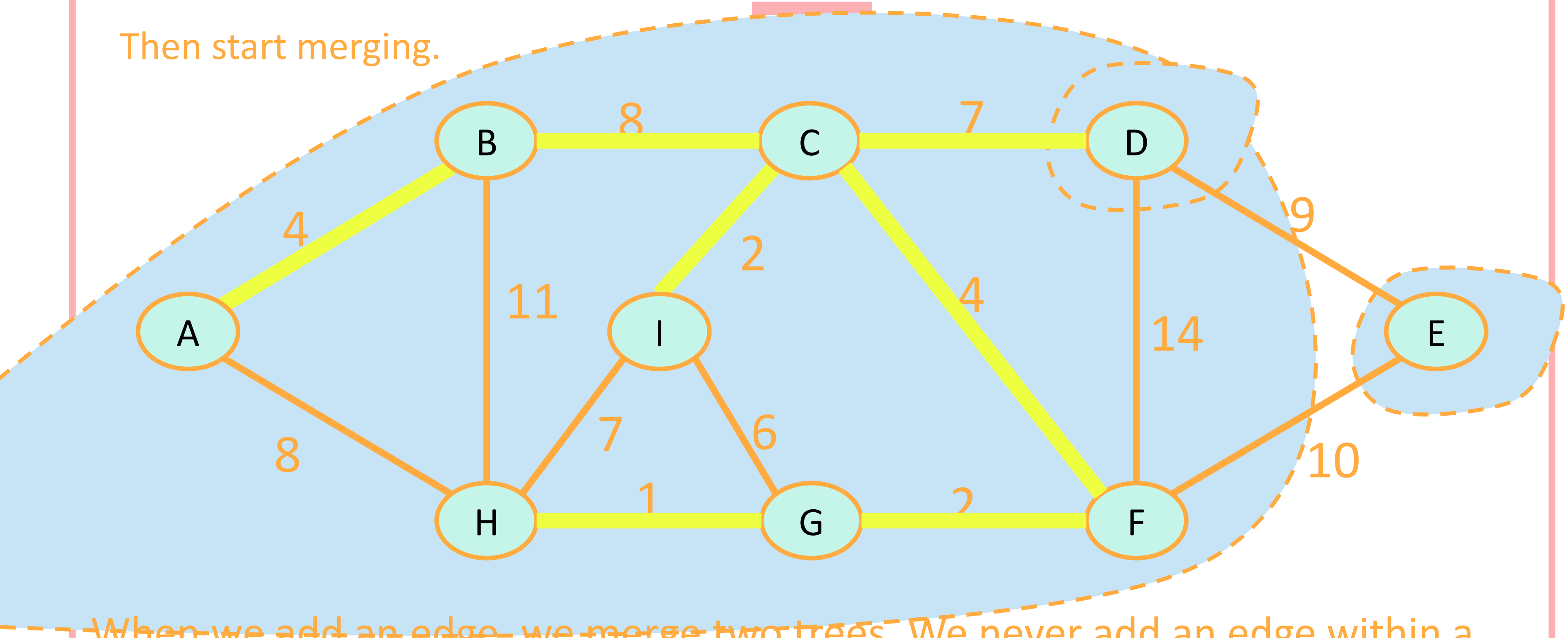
Then start merging



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

At each step of Kruskal's, we maintain a forest (collection of trees).

Then start merging.

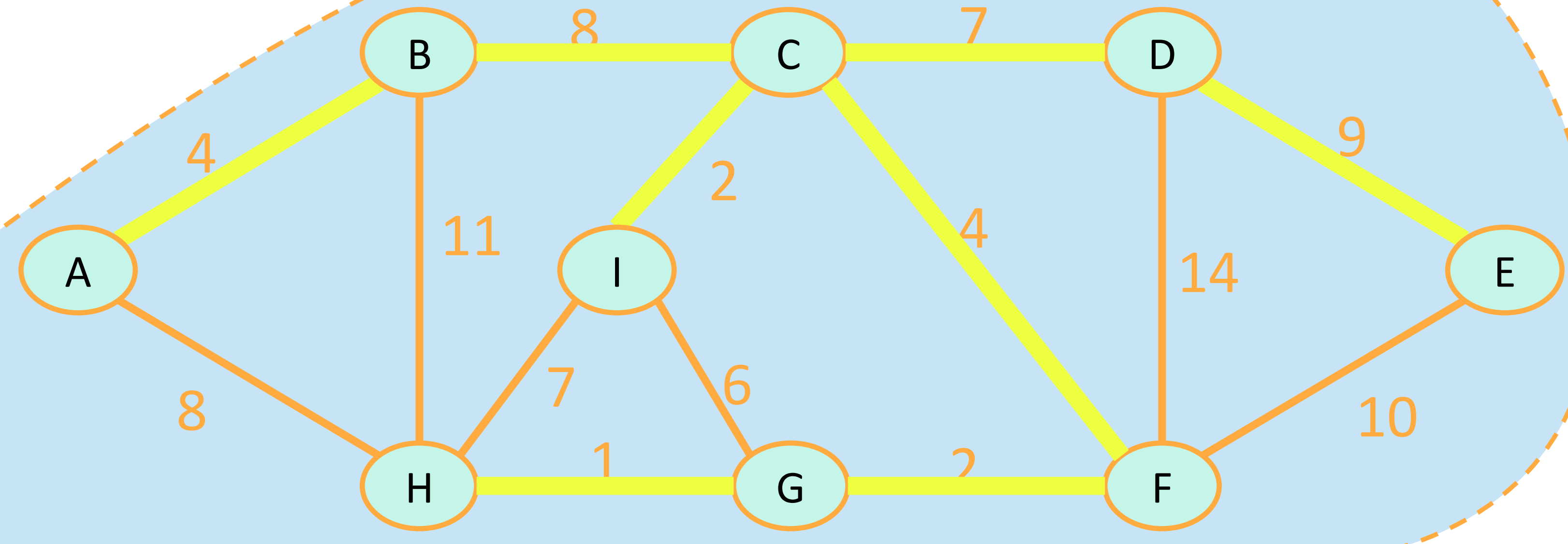


When we add an edge, we merge two trees. We never add an edge within a tree since that would create a cycle.

At each step of Kruskal's, we maintain a forest (collection of trees).

Stop when we have one big tree!

Then start merging.



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

- **Worst-case O(Elog(V))**

Kruskal Pseudocode (Take 2)

- **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

Compare and contrast

- Prim's
  - Looks like Dijkstra's
  - Explores nodes based on edges we can reach
  - Has a source node
  - MST so-far is always connected
- Kruskal's
  - Explore edges greedily instead of nodes
  - No start node
  - MST can be disconnected and slowly become connected as the algorithm runs.

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

COMP - 285

Analysis of Algorithms

Welcome to COMP 285

Lecture 17: Minimum Spanning Trees

Lecturer: Chris Lucas (cflucas@ncat.edu)