MINIMUM SPANNING TREES: PRIM-JARNIK'S ALGORITHM KRUSKAL'S ALGORITHM

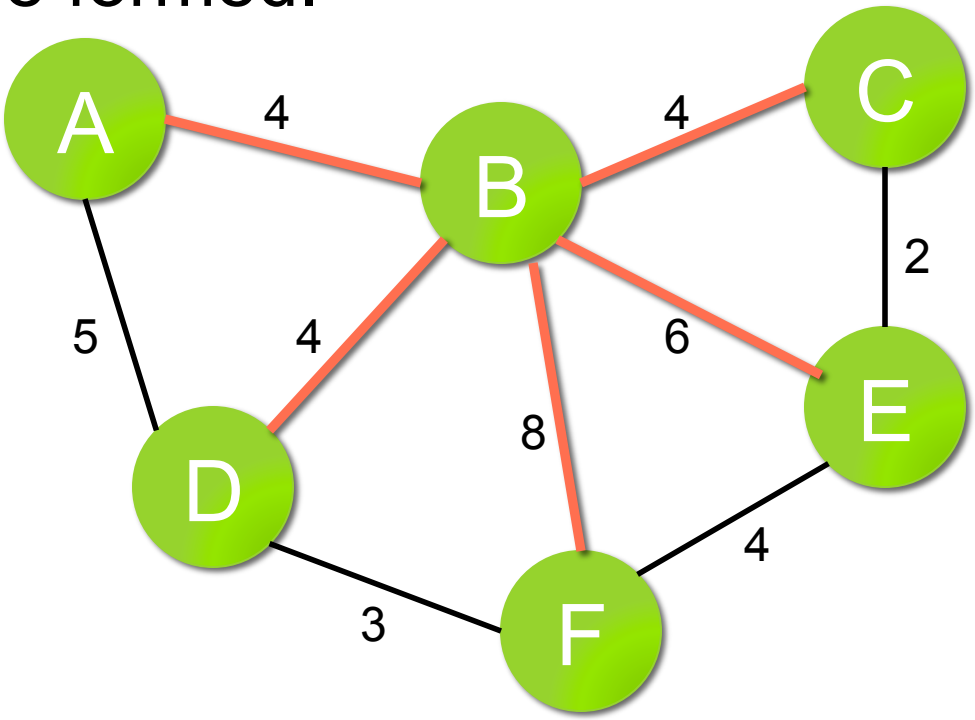
CS16: Introduction to Data Structures & Algorithms

Outline

1. Minimum Spanning Trees
2. Prim-Jarnik's Algorithm
3. Prim-Jarnik Analysis
4. Prim-Jarnik Proof of Correctness
5. Kruskal's Algorithm
6. Union-Find
7. Kruskal Analysis
8. Kruskal Proof of Correctness

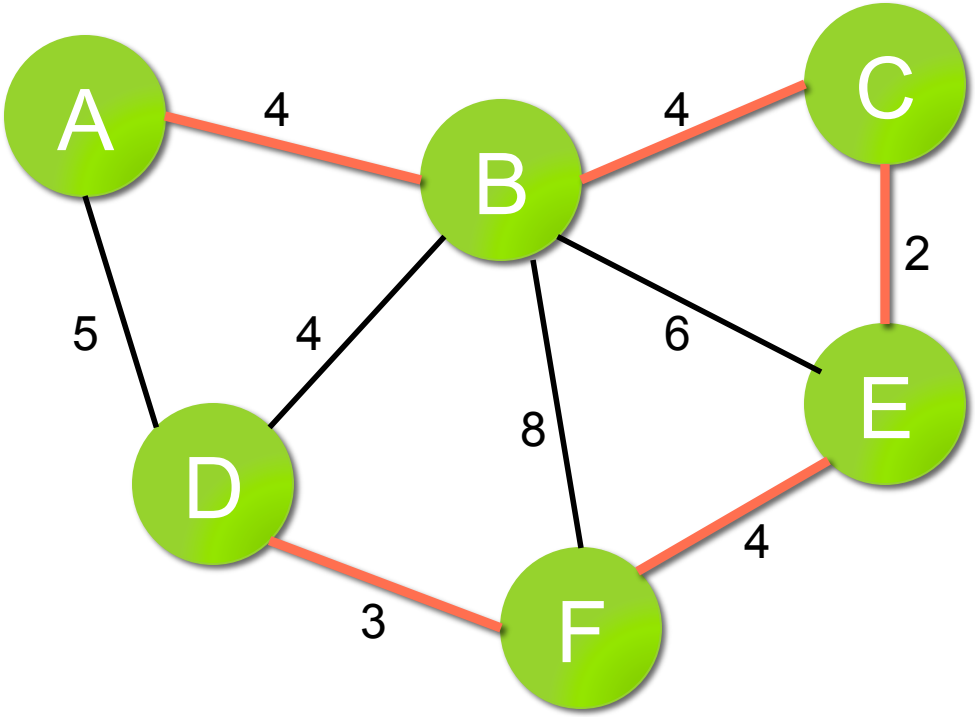
Spanning Trees

- A *spanning tree* of a graph is a selection of edges of the graph that form a tree spanning every vertex. That is, every vertex in the tree, but no cycles are formed.



Minimum Spanning Trees

- A *minimum spanning tree* (MST) is a spanning tree of a weighted graph with minimum total edge weight



Applications

- **Circuit design**
  - Necessary to wire pins together in order to make them electrically equivalent
  - An MST will represent the connections between these pins that requires the least amount of wire
- **Telephone substations**
  - All need to be connected
  - Use an MST to connect them all using the least amount of wire to save money

The Problem

Billy Mays seeks to connect his chain of tropical islands with bridges, but being a frugal man, he wants to use the least amount of material possible…

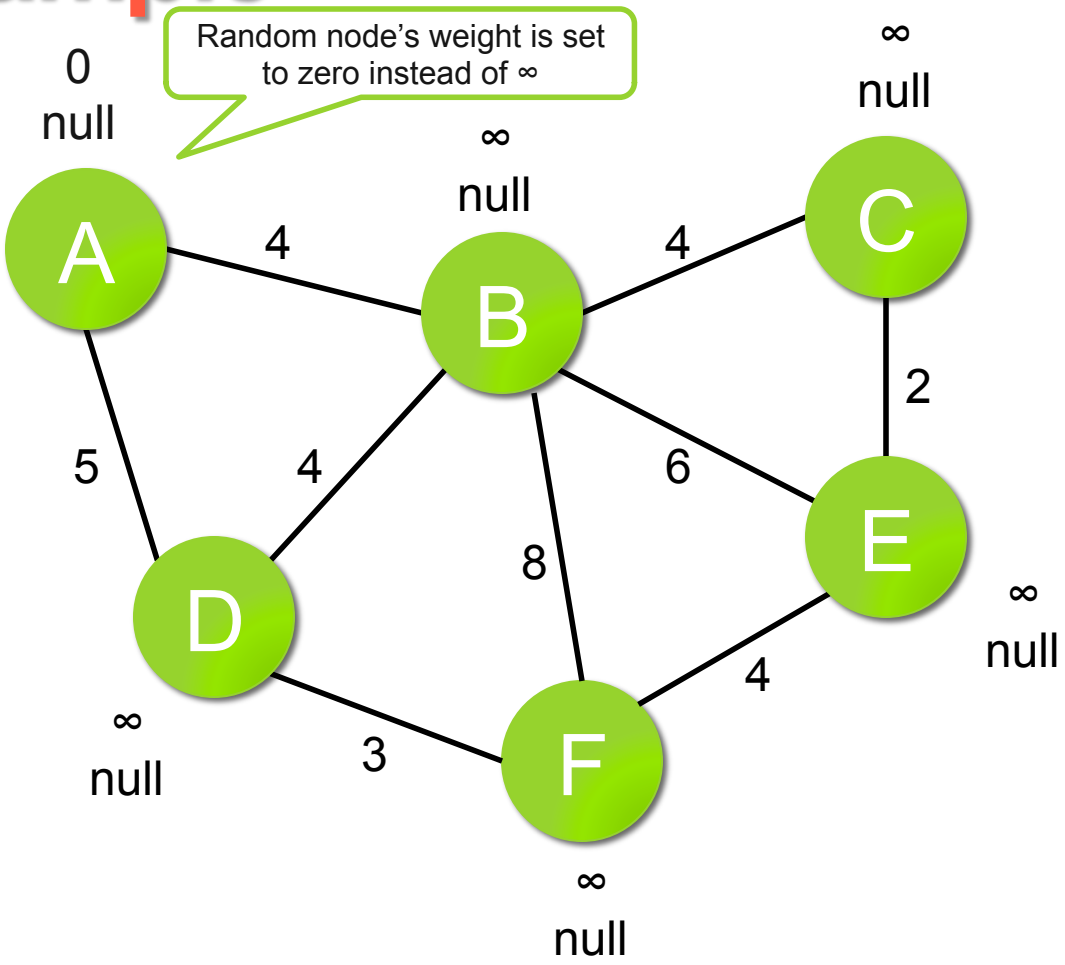
How do we find an MST?

- Two algorithms to find the MST of an undirected, weighted graph
  - Prim-Jarnik's Algorithm
  - Kruskal's Algorithm

Prim-Jarnik Algorithm

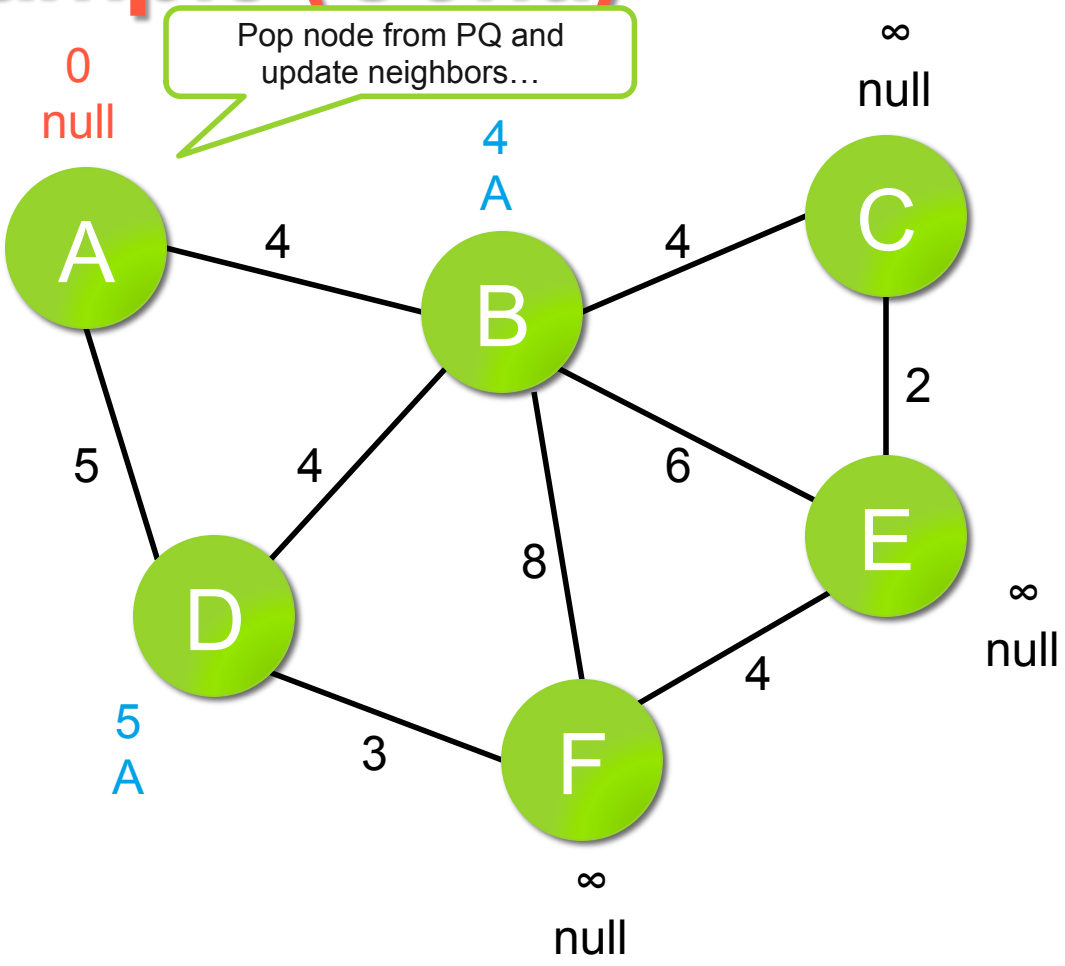
- Similar to Dijkstra's algorithm
- Traverse the graph starting at a random node
- Maintain a priority queue of nodes with the weight of the edge connecting them to the MST as their priorities
  - Unadded nodes start with infinite cost
- At every step:
  - Connect the node that has lowest cost
  - Update ("relax") the neighbors as necessary
- When all the nodes have been added to the MST, the algorithm terminates

Example



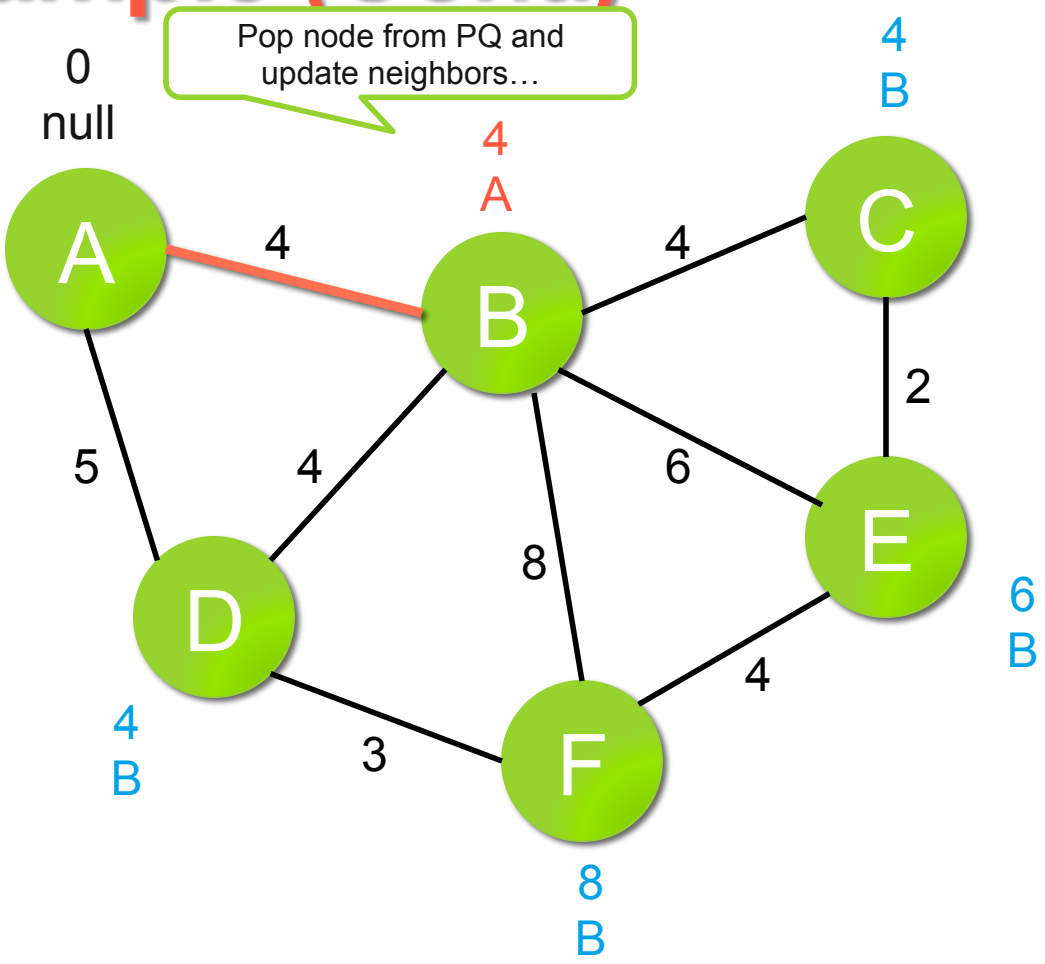
PQ = [(0,A),(∞,B),(∞,C),(∞,D),(∞,E),(∞,F)]

Example (Cont.)



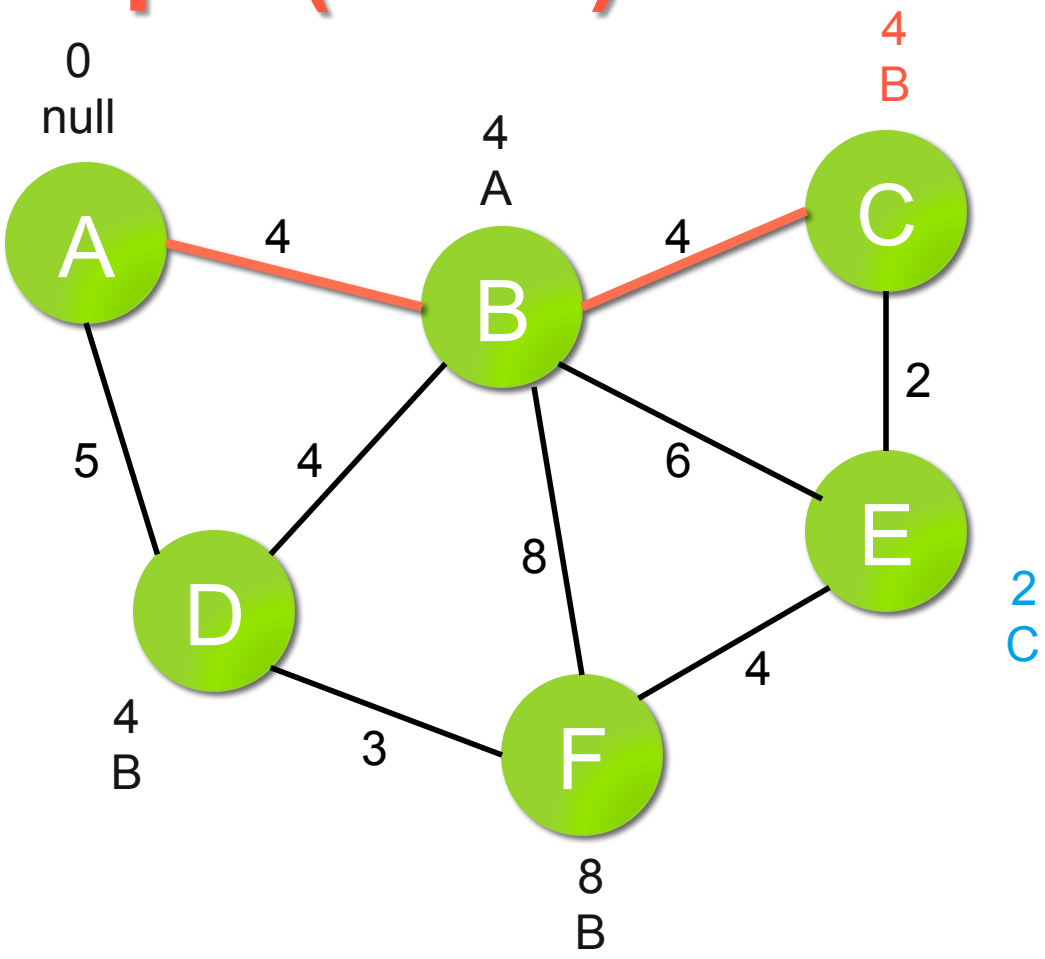
PQ = [(4,B),(5,D),(∞,C),(∞,E),(∞,F)]

Example (Cont.)



PQ = [(4,C),(4,D),(6,E),(8,F)]

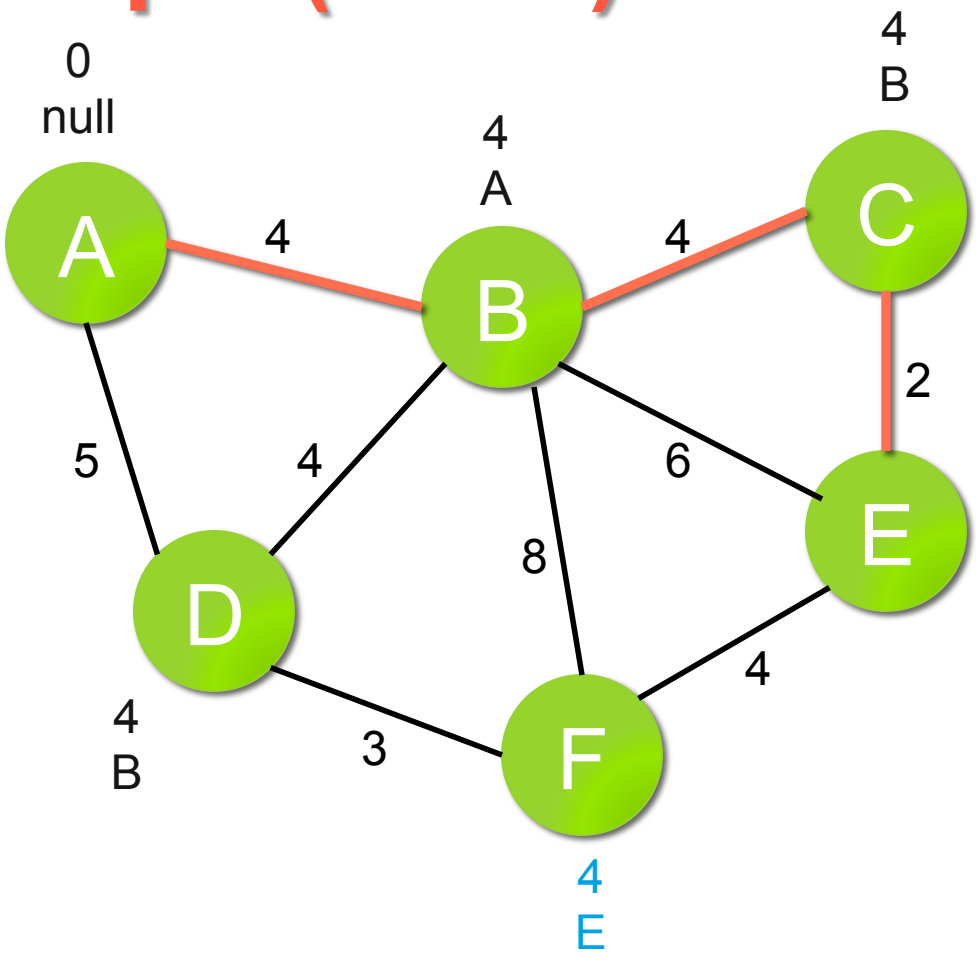
Example (Cont.)



Pop node from PQ and update neighbors…

PQ = [(2,E),(4,D),(8,F)]

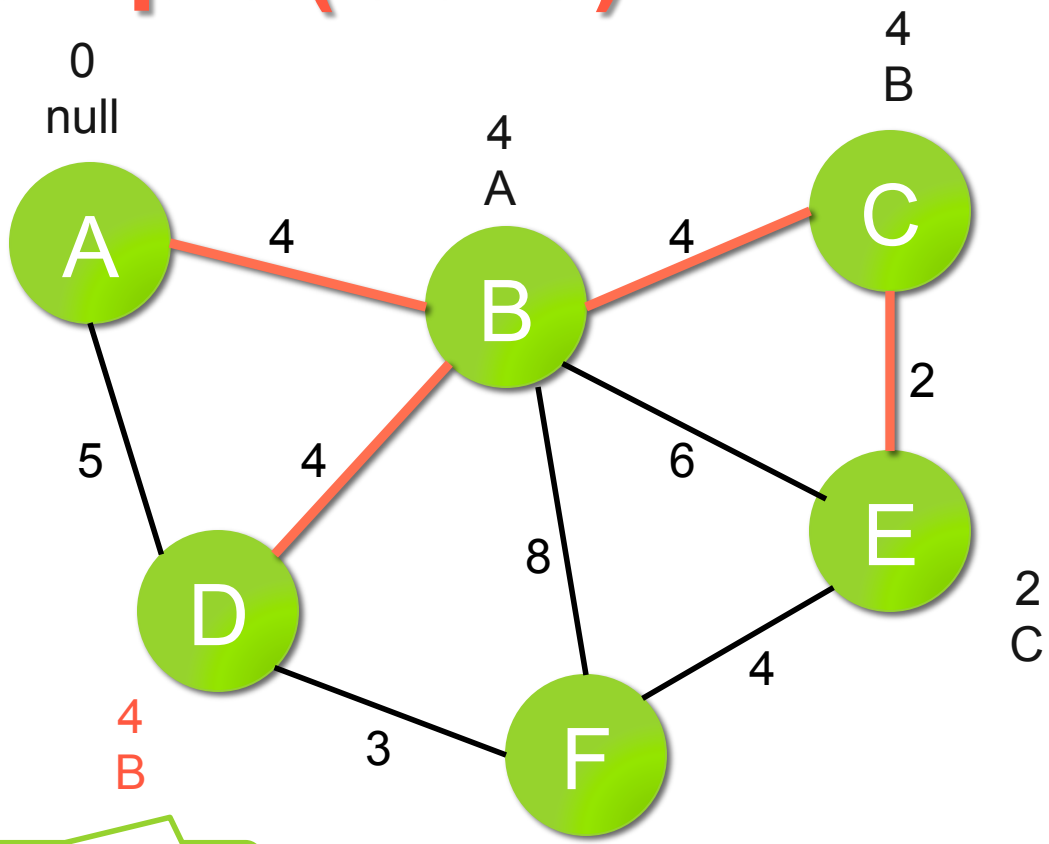
Example (Cont.)



Pop node from PQ and update neighbors…

PQ = [(4,D),(4,F)]

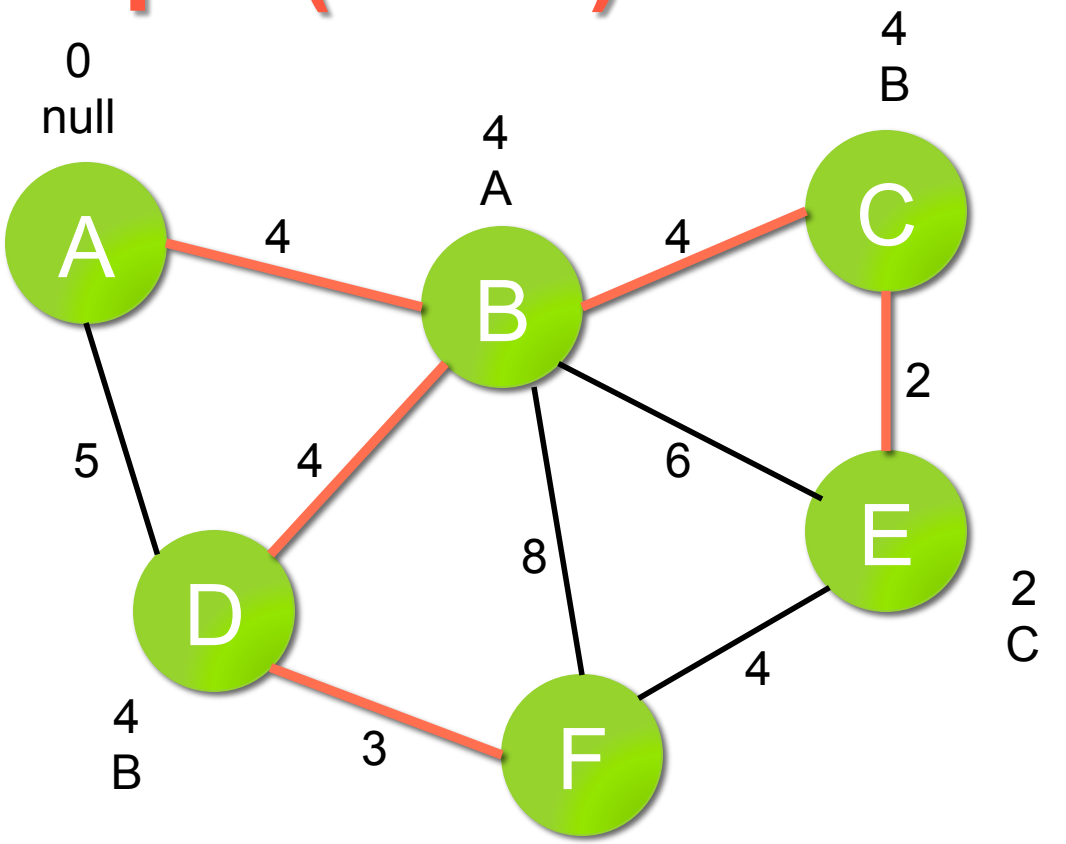
Example (Cont.)



Pop node from PQ and update neighbors…

PQ = [(3,F)]

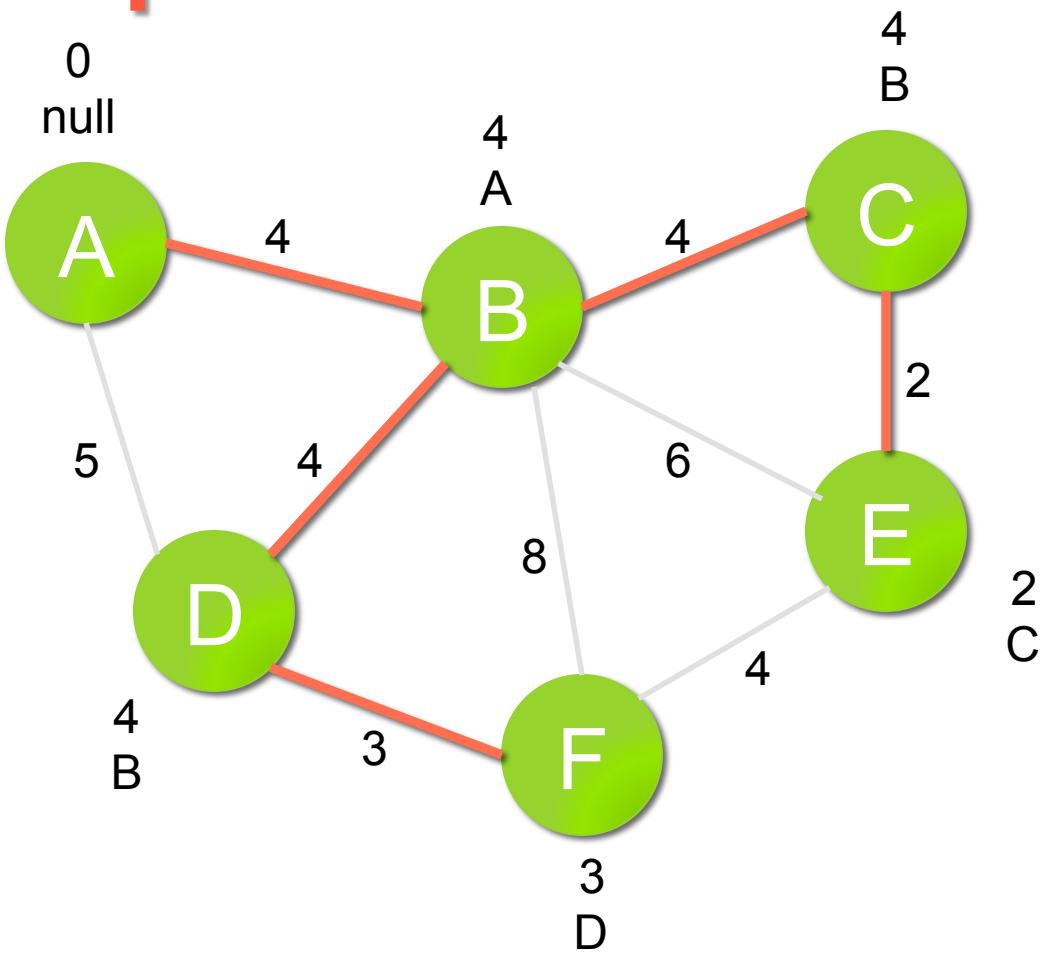
Example (Cont.)



Pop node from PQ and update neighbors…

PQ = [ ]

Example: Finished



Pseudocode

```
function prim(G):
   // Input: weighted, undirected graph G with vertices V
   // Output: list of edges in MST
   for all v in V:
      v.cost = ∞
      v.prev = null
   source = a random v in V
   source.cost = 0
   MST = []
   PQ = PriorityQueue(V) // priorities will be v.cost values
   while PQ is not empty:
      v = PQ.removeMin()
      if v.prev != null:
         MST.append((v, v.prev))
      for all incident edges (v,u) of v:
         if u.cost > (v,u).weight:
            u.cost = (v,u).weight
            u.prev = v
            PQ.replaceKey(u, u.cost)

  return MST
```

Runtime Analysis

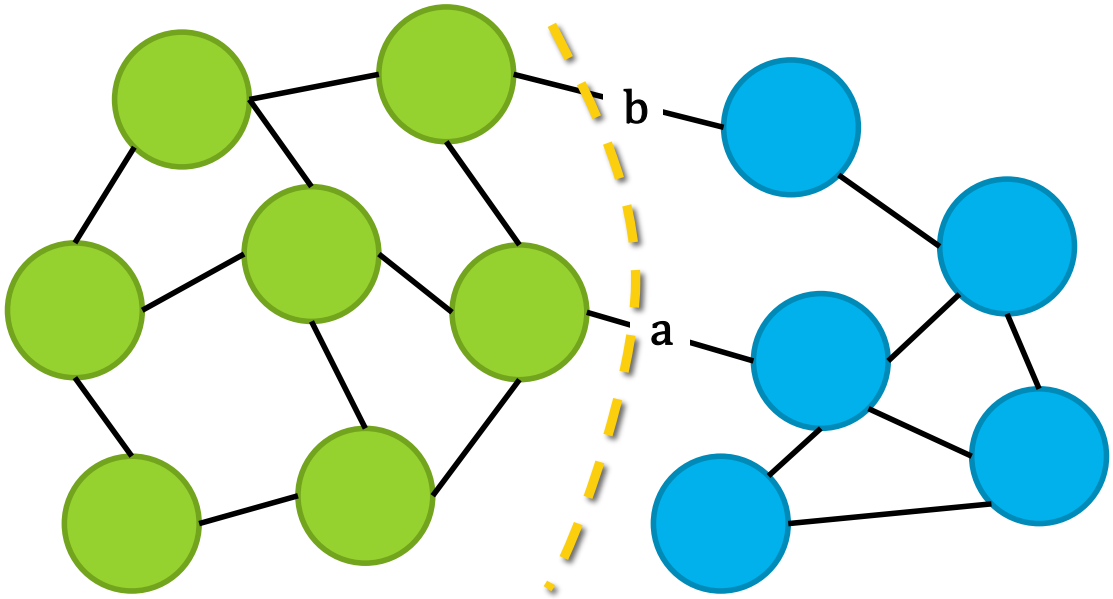
- Decorating nodes with distance and previous pointers: $O(|V|)$
- Putting nodes into priority queue with distance as key: $O(|V| \log |V|)$ (really $O(|V|)$ since infinite priorities)
- Within the while loop:
  - We will look at each vertex once when we delete the min from the priority queue: $O(|V| \log |V|)$
  - We look at each edge twice: once when we're looking at its to vertex, and the second time when we're looking at its from vertex. Replacing the key in the priority queue is $\log |V|$ time, thus $O(|E| \log |V|)$
- Total Runtime:
  - $O(|V| + |V| \log |V| + |E| \log |V|) = \mathbf{O((|E| + |V|) \log |V|)}$

Proof of Correctness

- A common strategy used to prove greedy algorithms like Prim-Jarnik's is to show that the algorithm is "always correct" at every step
  - More formally, we prove an *invariant* that holds at every step of the algorithm. Then, if you apply that invariant to the very last step, it proves that the entire algorithm is correct!
  - Best way to do this is by induction
- The tricky part is coming up with the right invariant
  - "At every iteration, Prim-Jarnik's algorithm constructs an MST" would be a great invariant to have for our conclusion, but it's just not true for all the intermediate steps
  - Instead, we'll prove the statement "$P(n)$: there exists an MST that contains the first n edges added by Prim's algorithm"

Graph Cuts

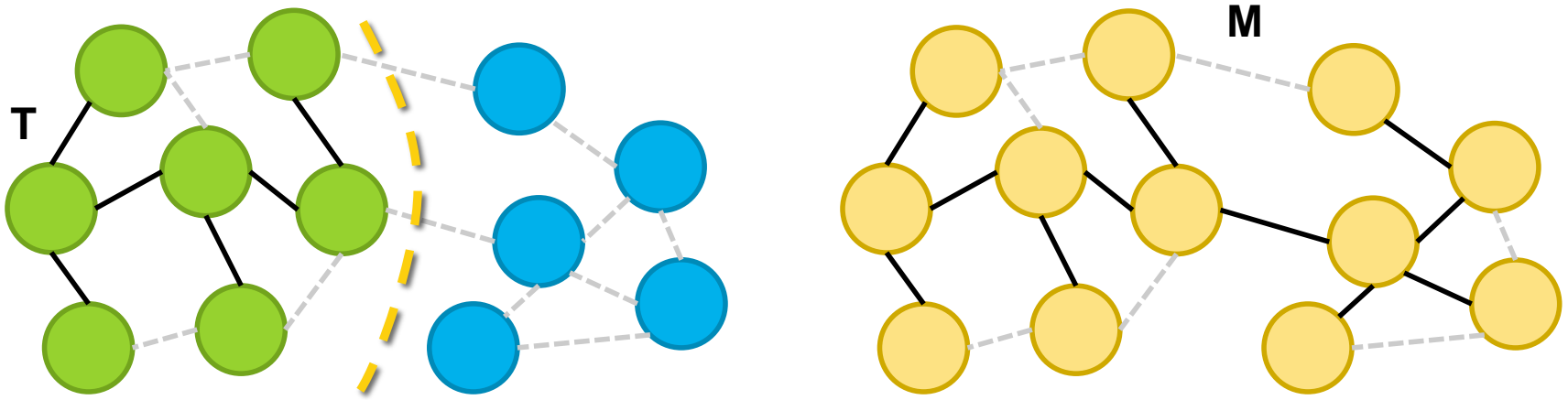
- It will help us in our proof to understand the idea of a *cut*, any partition of the vertices of a graph into two groups



- Here the graph has been partitioned into two groups, with edges *b* and *a* joining the two

Proof of Correctness (2)

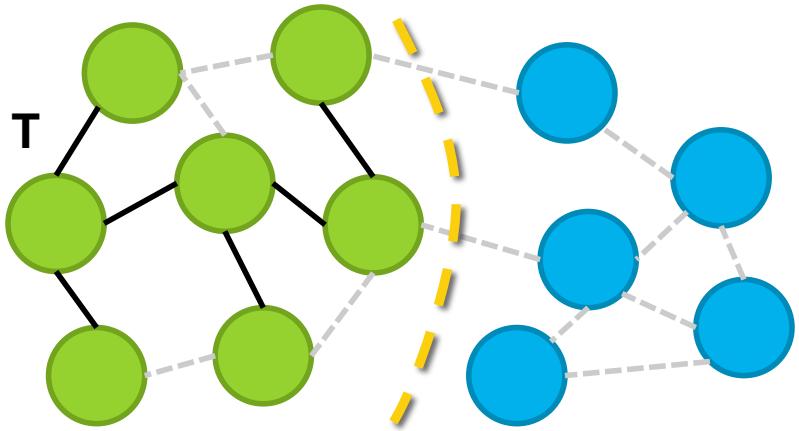
- **P(n):** The first n edges added by Prim's algorithm form a subtree of some MST
- **Base Case:** No edges have been added yet. P(0) is trivially true.
- **Inductive Hypothesis:**
  - Assume P(k): The first k edges added by Prim's algorithm form a tree **T** (the black edges between the green nodes), which is a subtree of some MST **M**



Proof of Correctness (3)

Inductive Step:

- Let **e** be the $(k+1)^{th}$ edge that is added by the algorithm
- **e** will connect **T** (the green nodes) to an unvisited node (one of the blue nodes)
- We want to show that adding **e** to **T** forms a subtree of some MST **M'** (which may or may not be the same MST as **M**)

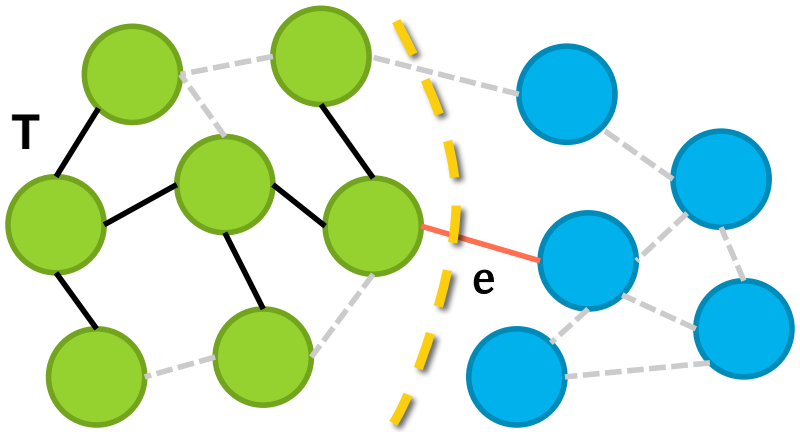


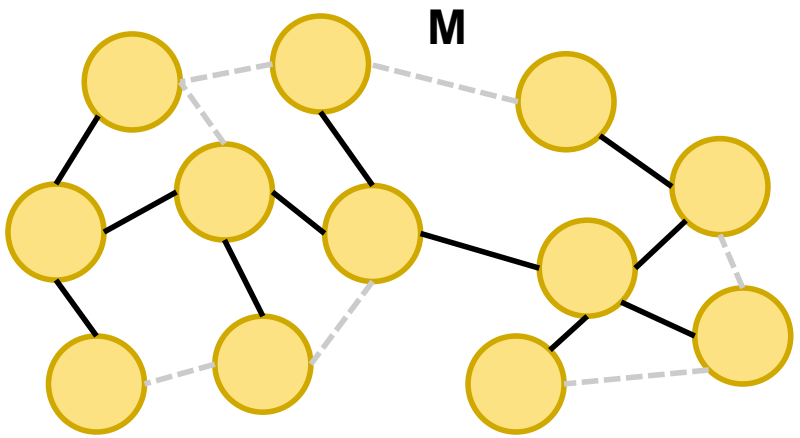
Proof of Correctness (4)

There are two cases:

1) **e** is in our original MST **M**
2) **e** is not in **M**

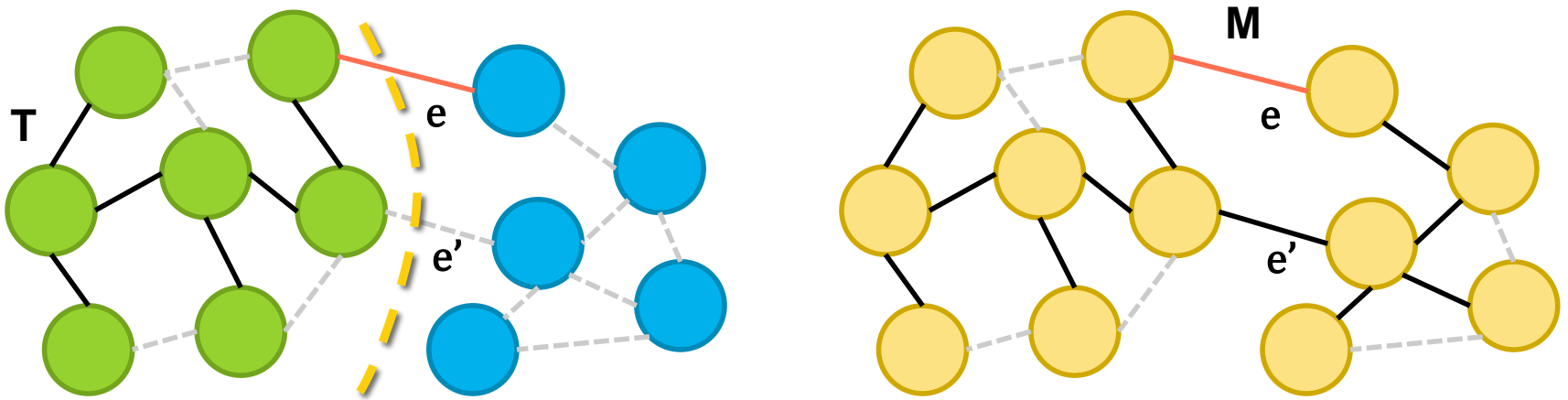
- Case 1: **e** is in **M**
  - Woo! There exists an MST (in this case **M**) that contains the first k+1 edges.
    - P(k+1) is true!





Proof of Correctness (5)

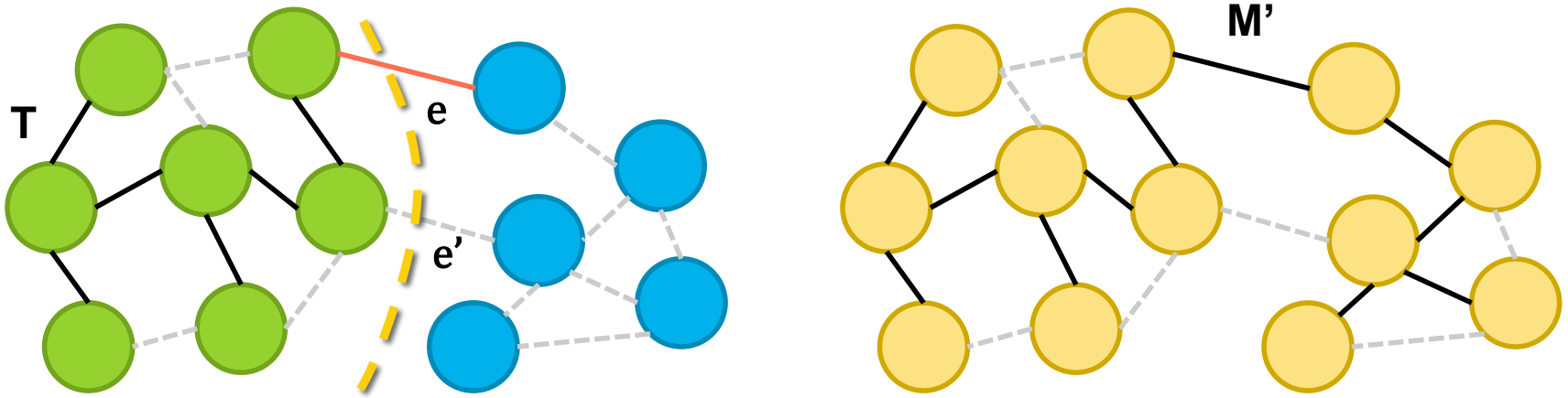
- Case 2: **e** is not in **M**
  - If we were to add **e** to **M**, this would create a cycle
    - Thus, there must be another edge **e'** that also connects the nodes in **T** to the unvisited part of the graph



- We know **e**.weight ≤ **e'**.weight because Prim's chose **e** first

Proof of Correctness (6)

- Thus, if we add **e** to **M** and remove **e'**, we get a new MST **M'** that is *no larger* than **M** was and contains **T** and **e**



- Since **M'** is an MST that contains the first k+1 edges added by Prim's, P(k+1) is true

Conclusion: Since we have shown P(0) and shown P(k)➔P(k+1) for both possible cases, we have proven that the first n edges added by Prim's algorithm form a subtree of some MST

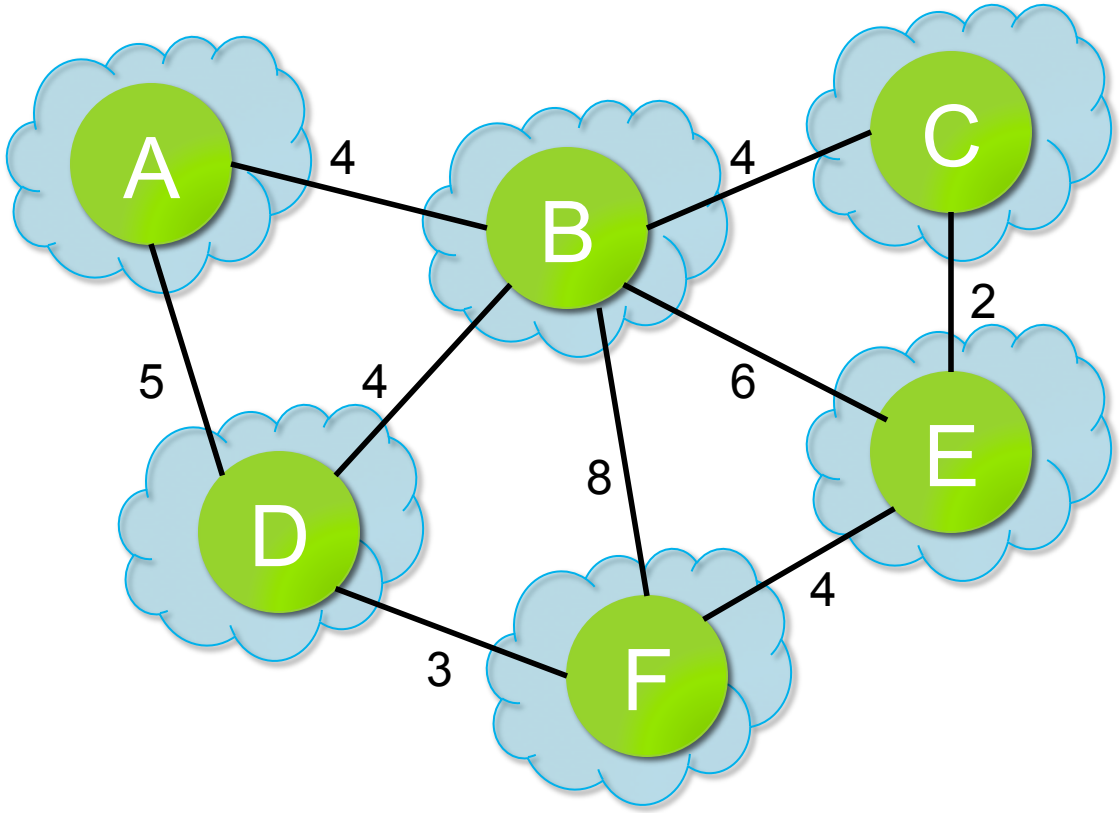
Kruskal's Algorithm

- Sort all edges of the graph by weight in ascending order.
- For each edge in the sorted list:
  - If adding the edge does not create a cycle, add it to the MST
- When you've gone through all the edges in the list, you're done!

Kruskal's Algorithm (2)

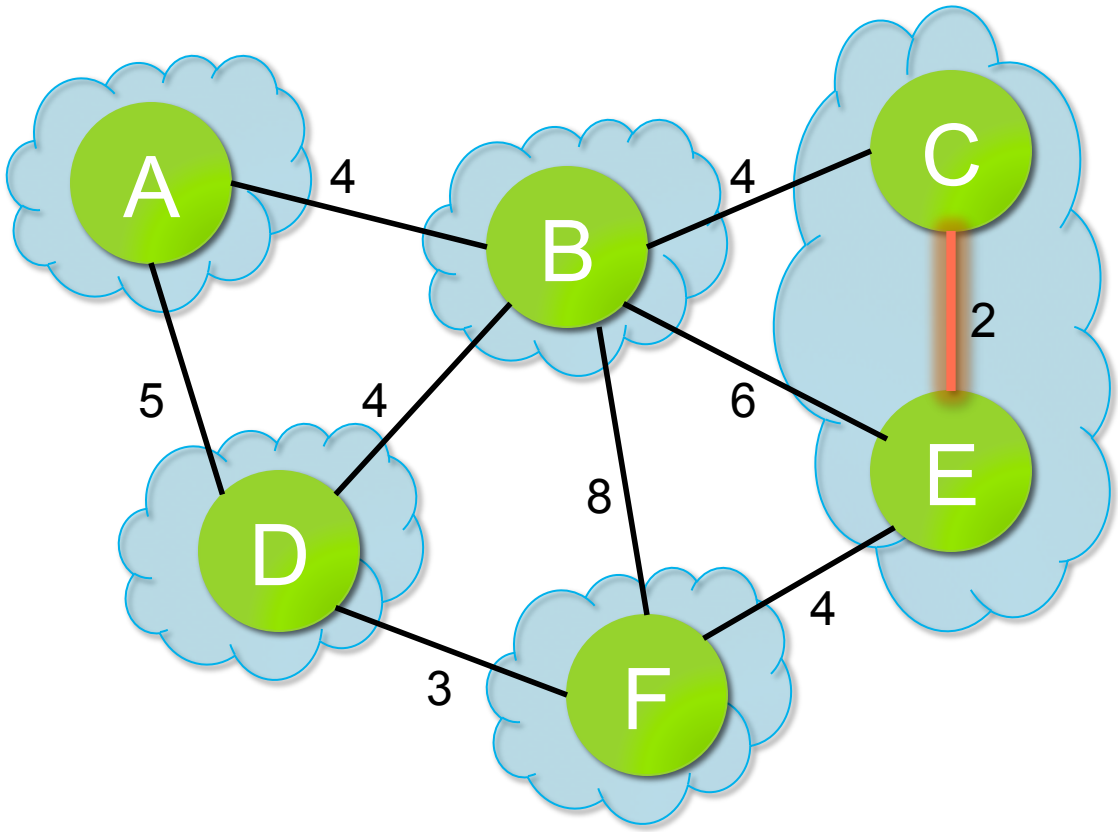
- How can we tell if adding an edge will create a cycle?
  - Could run BFS/DFS to detect a cycle, but that's so slow!
- Start by giving each vertex its own "cloud"
- When you add an edge to the MST, merge the clouds of the two endpoints
- If both endpoints of an edge are already in the same cloud, we know that adding that edge will create a cycle!

Example



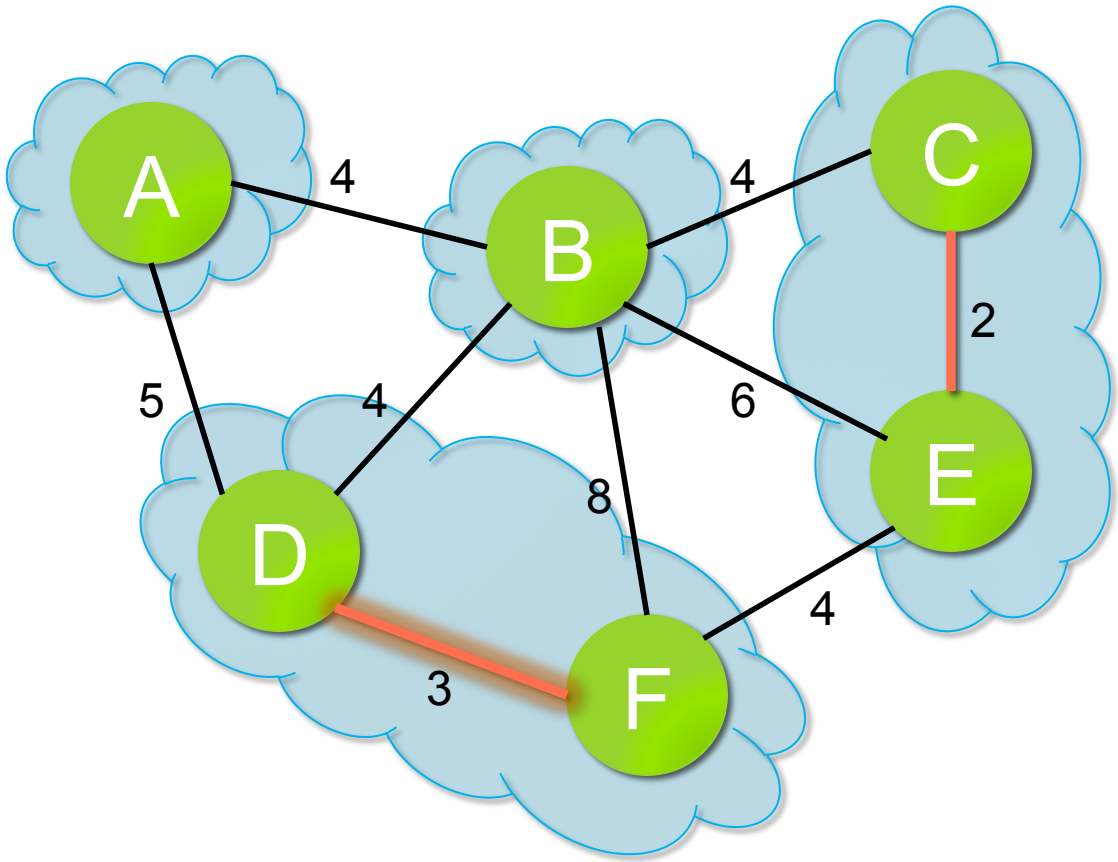
edges = [(C,E),(D,F),(B,C),(E,F),(B,D),(A,B),(A,D),(B,E),(B,F)]

Example (cont.)



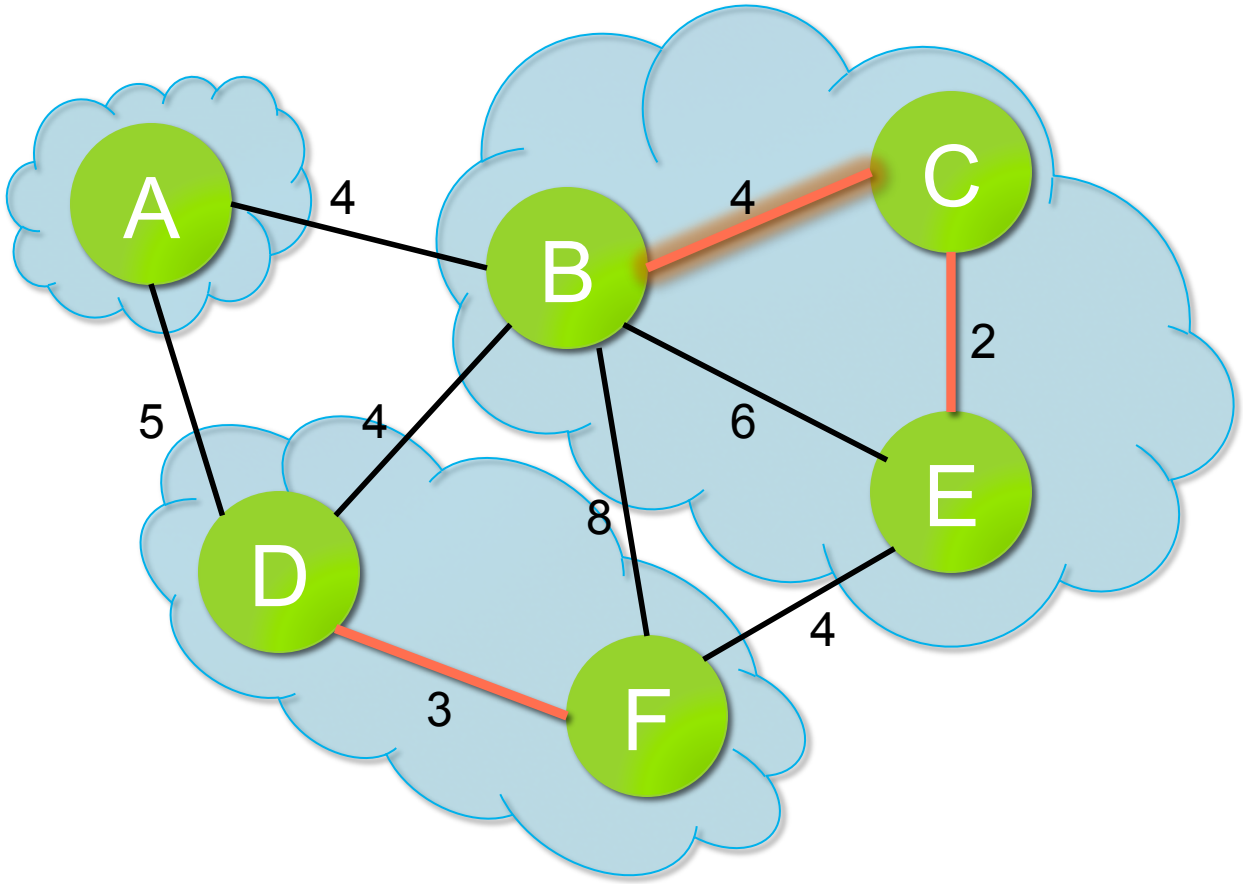
edges = [(D,F),(B,C),(E,F),(B,D),(A,B),(A,D),(B,E),(B,F)]

Example (cont.)



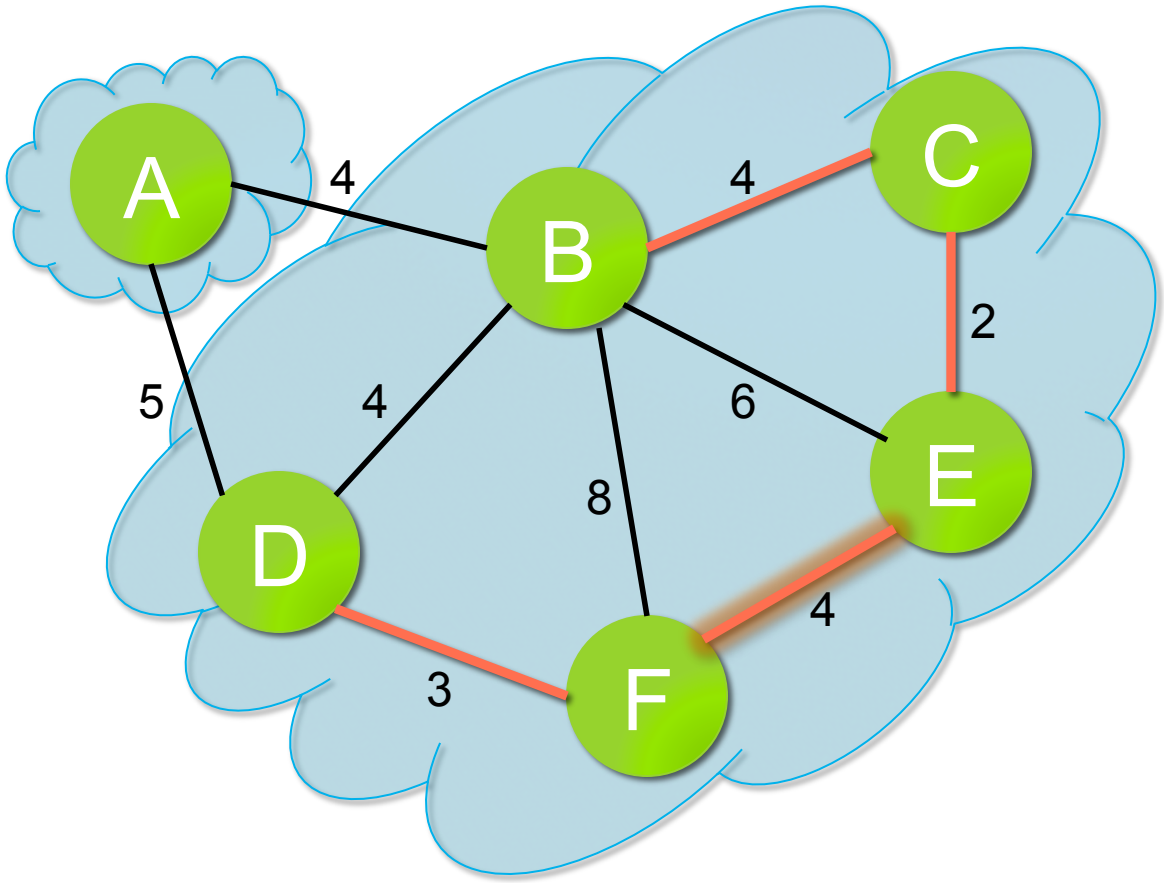
edges = [(B,C),(E,F),(B,D),(A,B),(A,D),(B,E),(B,F)]

Example (cont.)



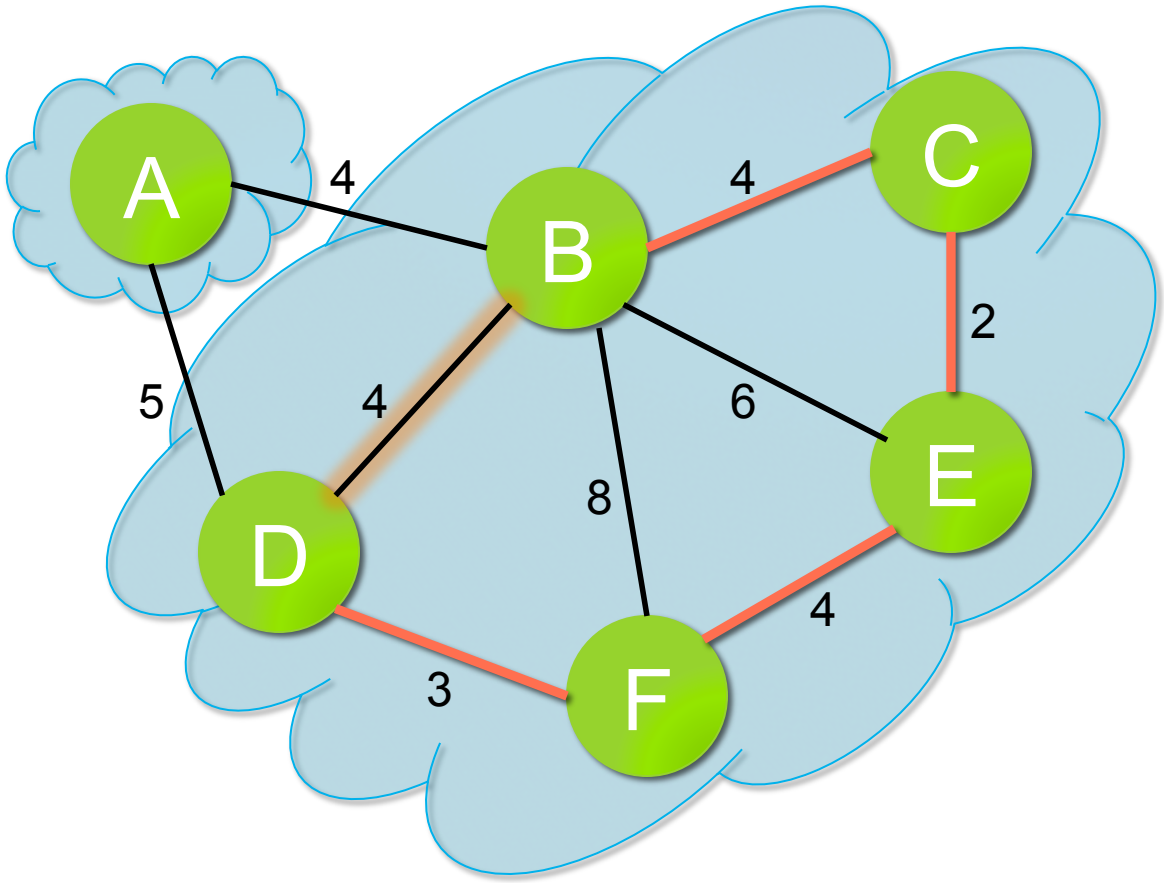
edges = [(E,F),(B,D),(A,B),(A,D),(B,E),(B,F)]

Example (cont.)



edges = [(B,D),(A,B),(A,D),(B,E),(B,F)]

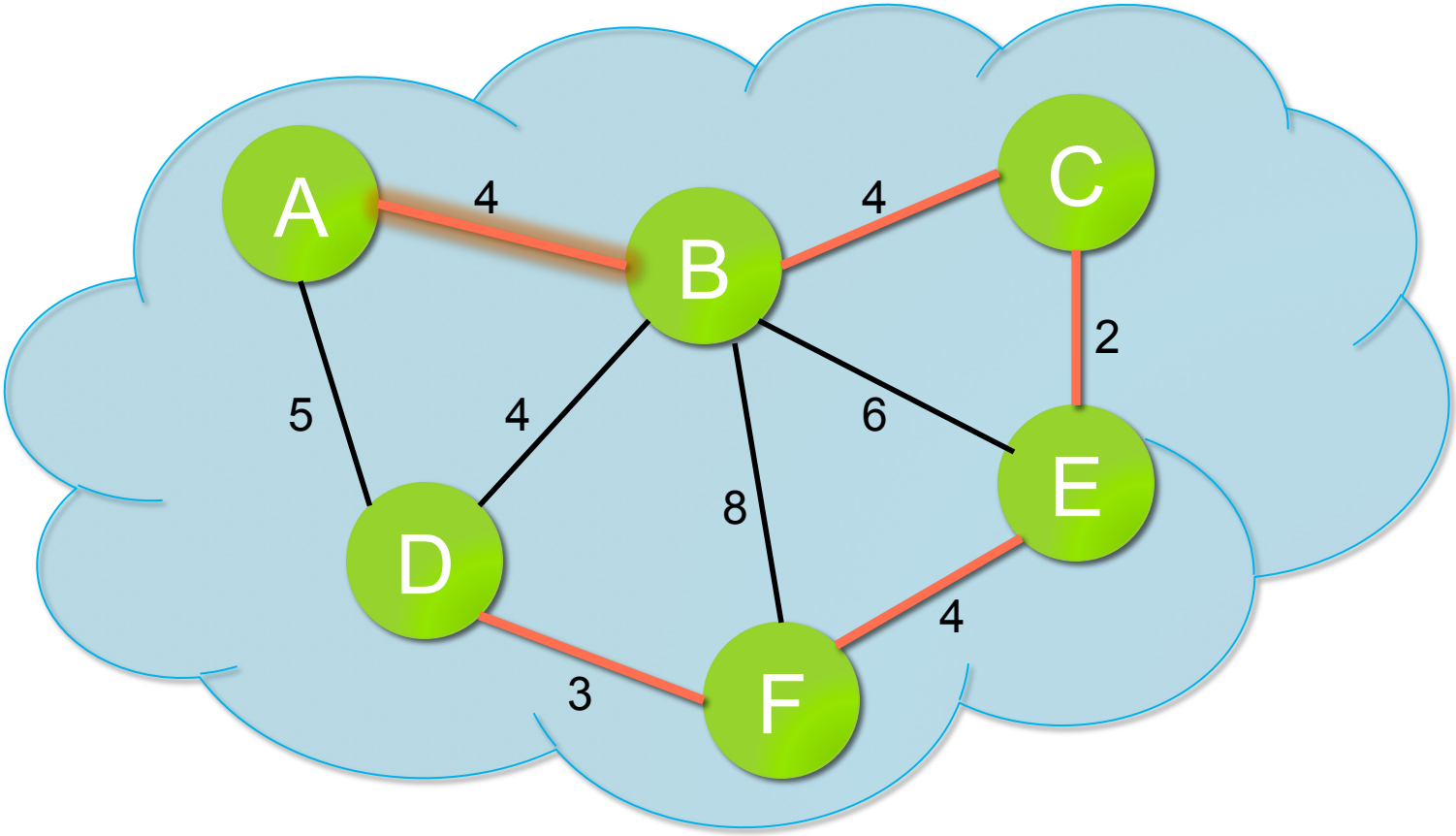
Example (cont.)



edges = [(A,B),(A,D),(B,E),(B,F)]

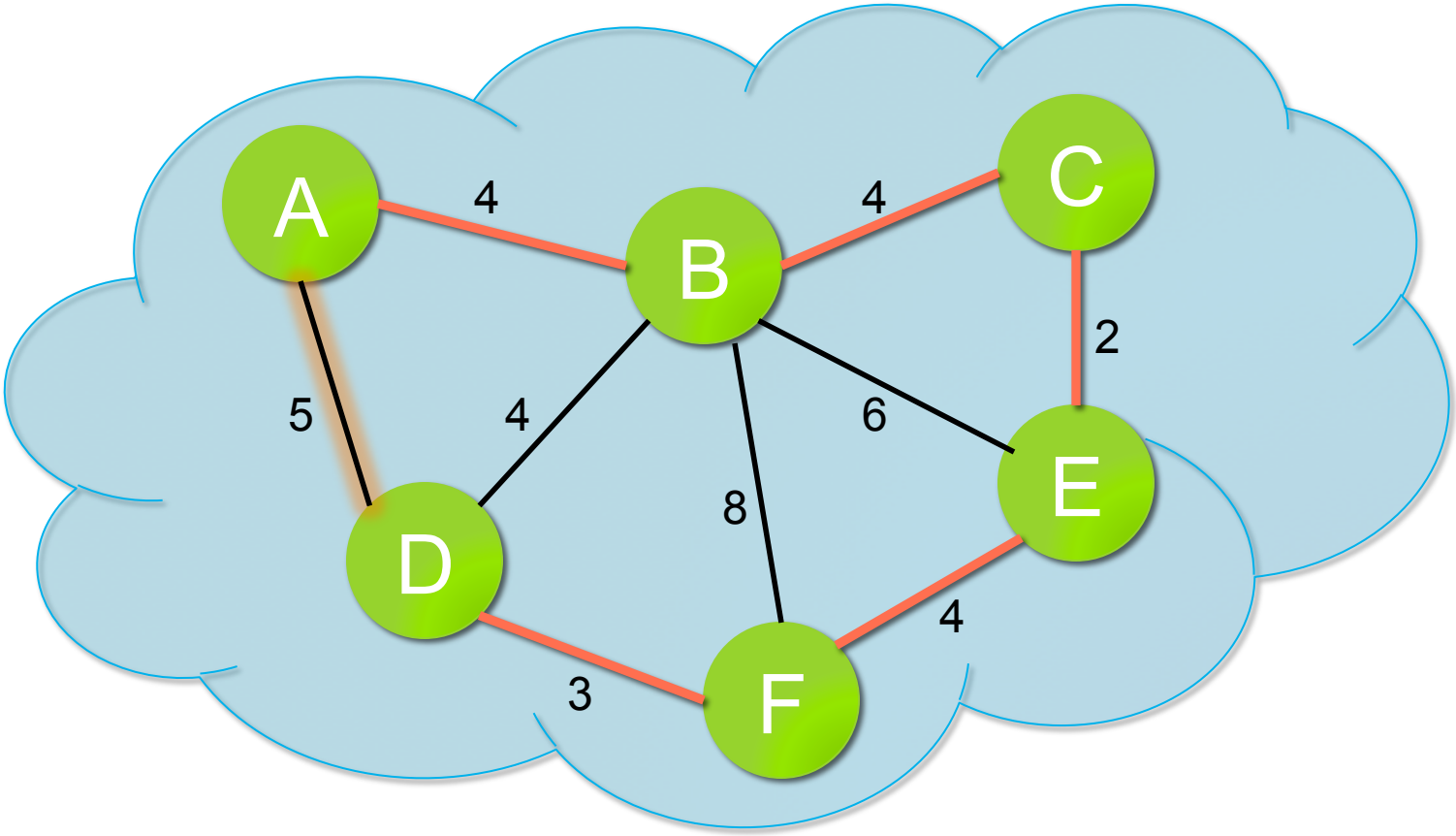
BD cannot be added to the MST because it would lead to a cycle!

Example (cont.)



edges = [(A,D),(B,E),(B,F)]

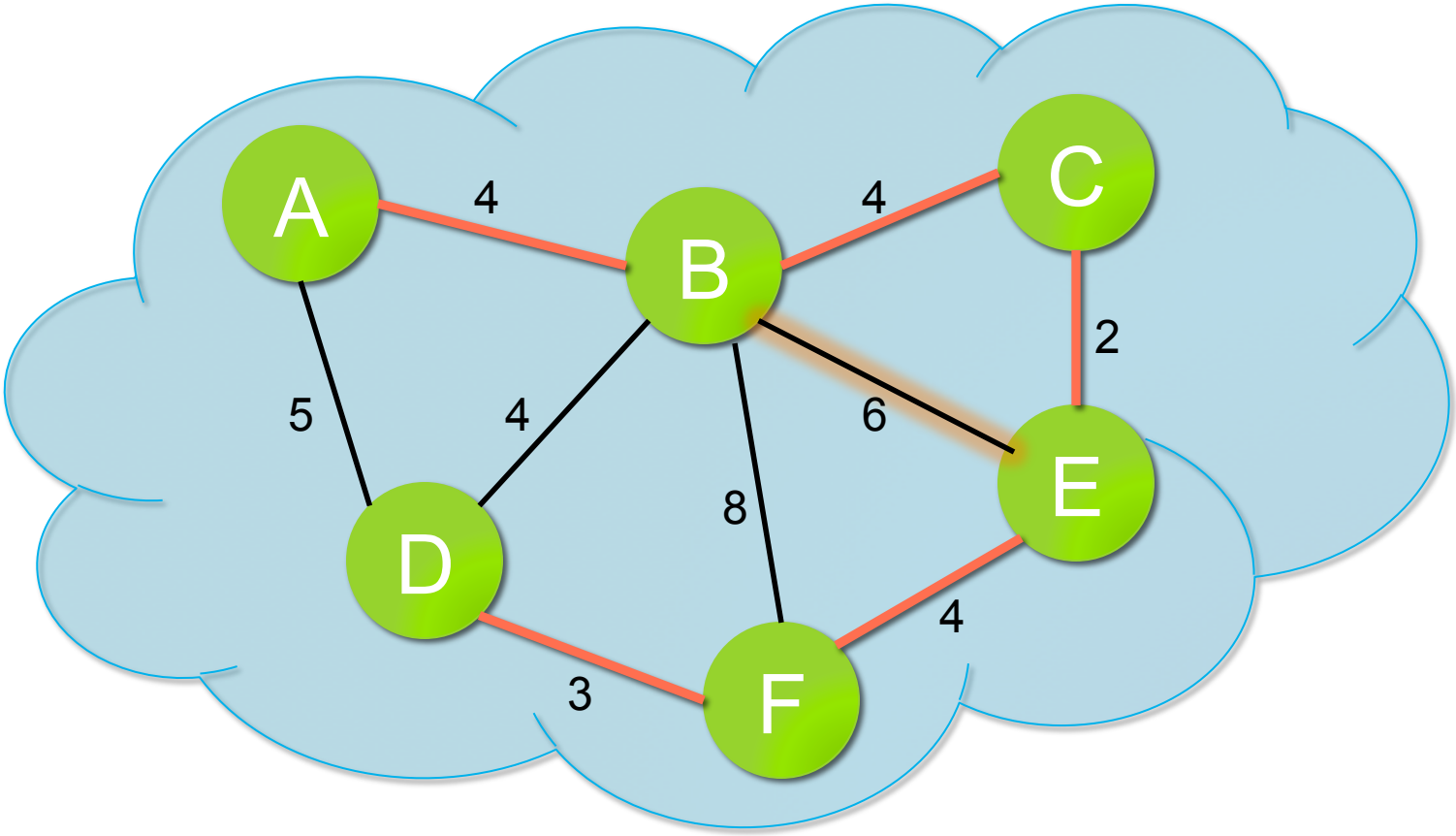
Example (cont.)



edges = [(B,E),(B,F)]

AD cannot be added to the MST because it would lead to a cycle!

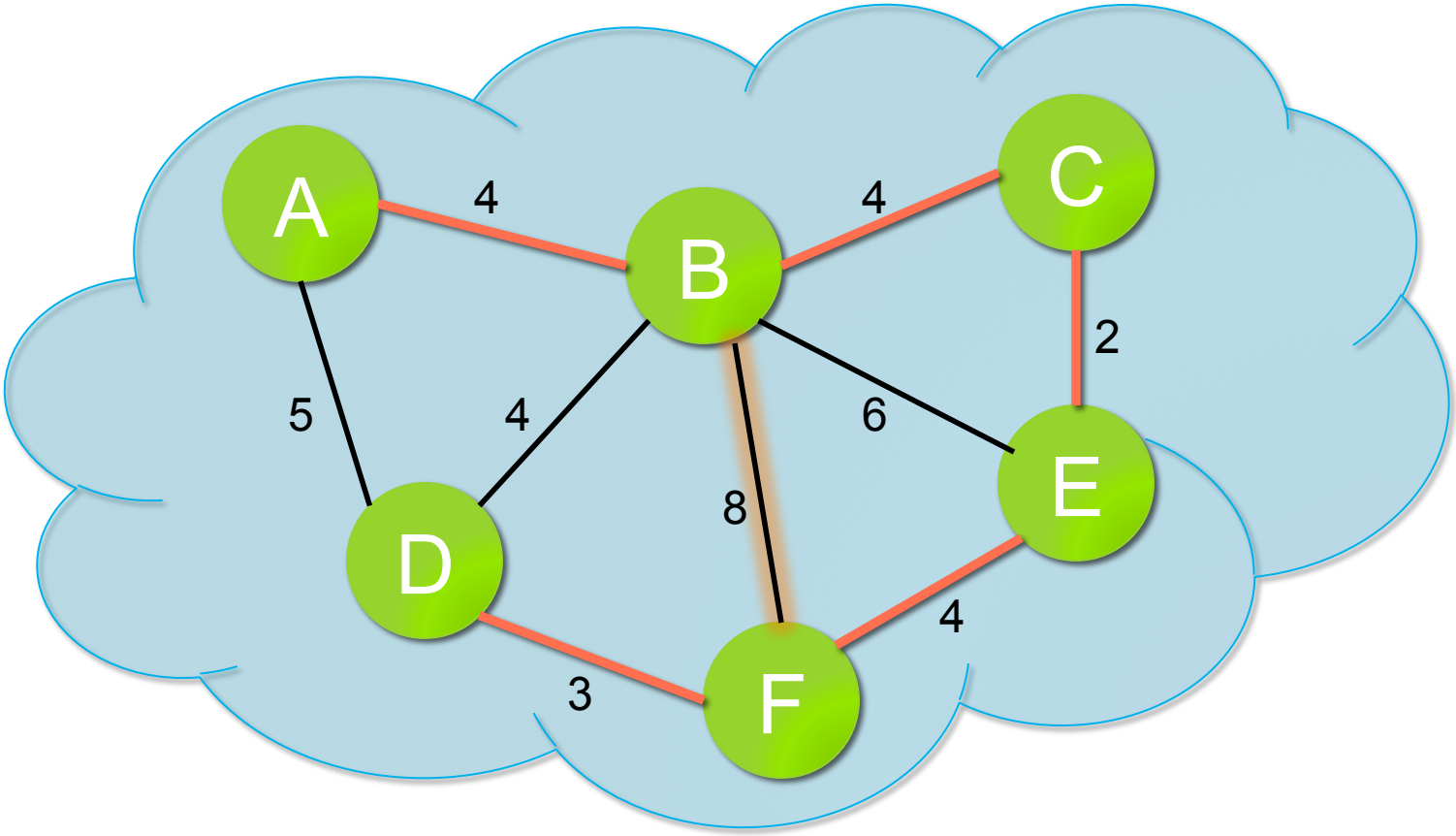
Example (cont.)



edges = [(B,F)]

BE cannot be added to the MST because it would lead to a cycle!

Example (cont.)



edges = [ ]

BF cannot be added to the MST because it would lead to a cycle!

Pseudocode

```
function kruskal(G):
    //Input: undirected, weighted graph G
    //Output: list of edges in MST
    for vertices v in G:
        makeCloud(v) //put every vertex into it own set
    MST = []
    for all edges (u,v) in G sorted by weight:
        if u and v are not in same cloud:
            add (u,v) to MST
            merge clouds containing u and v
    return MST
```

Merging Clouds – The Naïve Way

- This way involves assigning each vertex a different number to represent its initial "cloud"
- To merge clouds of vertices u and v, we will:
  - Find all the vertices in each cloud
  - Figure out which of the clouds is smaller
  - Redecorate all vertices in the smaller cloud with the bigger cloud's number

Merging Clouds – The Naïve Way

- Runtime: $O(|V|)$
  - Finding all vertices in u's and v's clouds is $O(|V|)$, because we will have to iterate through each vertex and check whether its cloud number matches u's or v's cloud number
  - Figuring out the smallest cloud is $O(1)$ as long as we keep track of the size of the clouds as we find the vertices in them
  - Changing the cloud numbers of the vertices in the smaller cloud is worst-case $O(|V|)$, because the cloud could be as big as |V|/2 vertices
  - Total Runtime: $O(|V|) + O(1) + O(|V|) = O(|V|)$

Runtime Analysis of Kruskal's with naïve union-find

```
function kruskal(G):
 //Input: undirected, weighted graph G
 //Output: list of edges in MST
 for vertices v in G:
      makeCloud(v)          O(|V|)
 MST = []
 O(|E|log|E|) for sorting edges
 for all edges (u,v) in G sorted by weight: O(|E|)
    if u and v are not in same cloud:
        add (u,v) to MST
        merge clouds containing u and v   O(|V|)
 return MST
```

Runtime

- Total Runtime:
  - $O(|V|)$ for iterating through vertices
  - $O(|E| \log|E|)$ for sorting edges
  - $O(|E|) * O(|V|)$ for iterating through edges and merging clouds naively.
- $O(|V|) + O(|E| \log|E|) + O(|E|) * O(|V|)$

$$= O(|V|\ |E|) = O(|V|^3)$$

- Can we do better?

Union-Find

- Merging clouds is expensive – on merging two clouds, you have to relabel all vertices in one cloud with the other cloud's number.
- Let's rethink this notion of clouds: instead of labeling every vertex with its cloud number, think of clouds as small trees.
- Every vertex in these trees has
  - a parent pointer, which leads up to the root of the tree
  - a rank, which measures how deep the tree is
- Simply put, the root of the tree will know to which cloud it belongs, while other vertices will need to ask the root what cloud it belongs to in order to determine their own clouds.

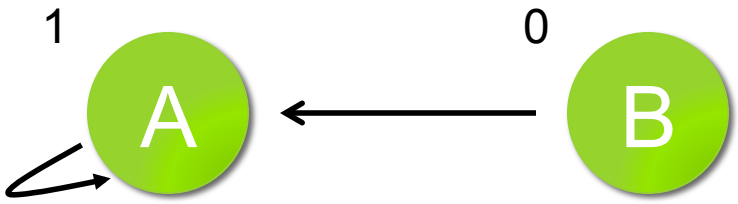
Implementing Union-Find

- At the start of Kruskal's algorithm, all vertices are put in their own clouds.

```
// Decorates every vertex with its parent
// pointer and rank order
function makeCloud(x):
    x.parent = x
    x.rank = 0
```

Implementing Union-Find

- At every step, we process one edge at a time and add it to our MST. So after every step, we combine two vertices into one cloud.
- Suppose A is in cloud 1 and B is in cloud 2. Instead of relabeling vertex B as cloud 1, simply make vertex B "point" to vertex A. Think of this as the "union" of two clouds.



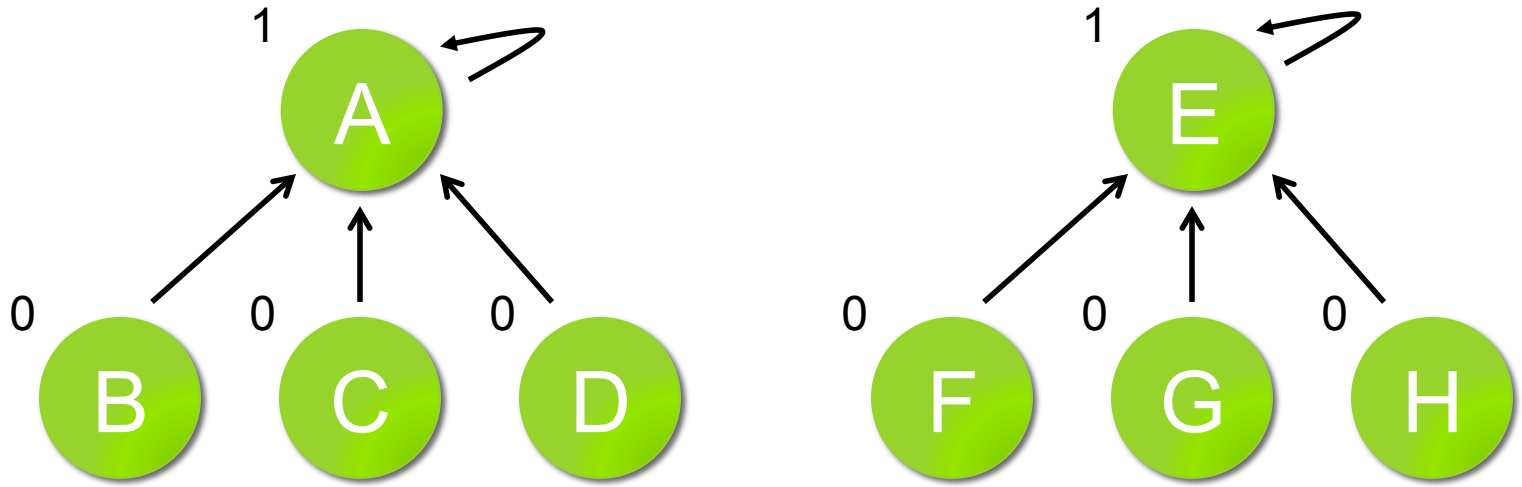
- Question: Given two clouds, how do you determine which one should point to the other?

Implementing Union-Find

- Answer: Using the “rank” property!
- For clouds of size 1, the rank of the root of the cloud is 0.
- For all clouds of size greater than 1, rank is updated during a `union(…)` operation:
  - When merging two clouds of the same rank, arbitrarily make one root point to the other and increment the rank of the root of the new, merged cloud by 1
  - When merging two clouds of different rank, make the lower-ranked cloud’s root point to the higher-ranked cloud’s root. Do not change any ranks.

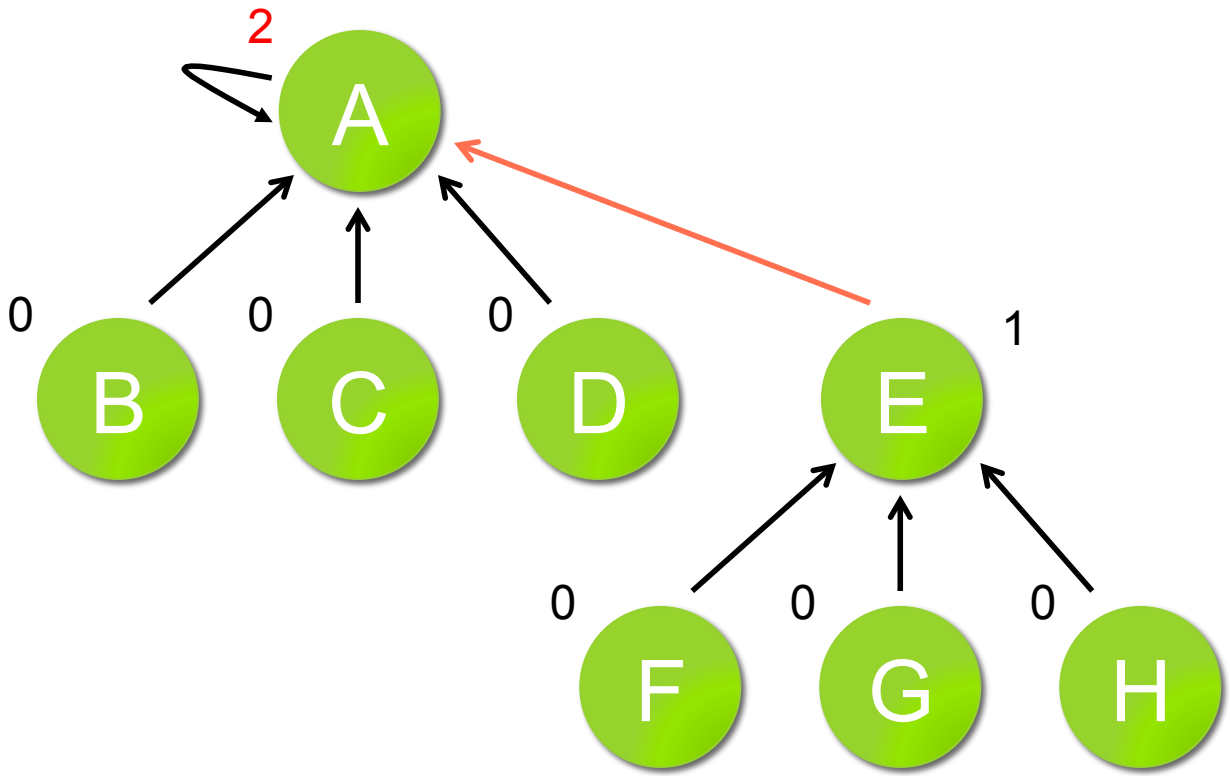
Implementing Union-Find

- Merging trees with same ranks:



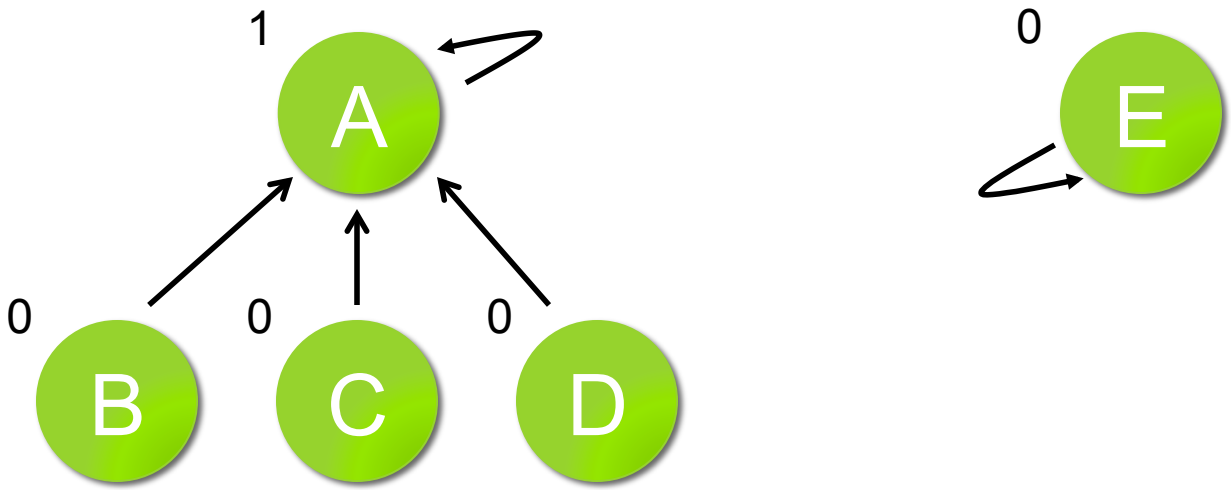
Implementing Union-Find

- Merging trees with same ranks:



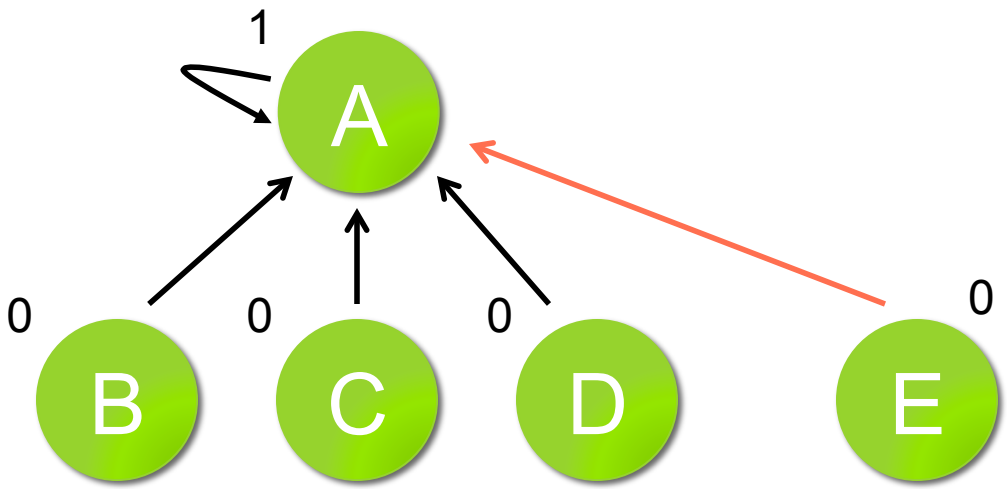
Implementing Union-Find

- Merging trees with different ranks:



Implementing Union-Find

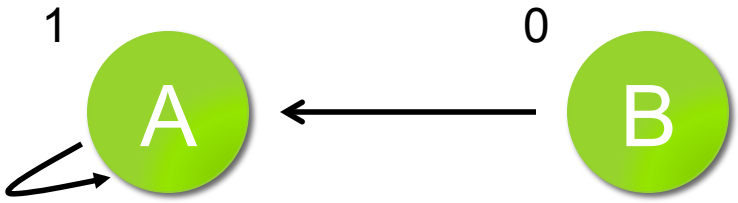
- Merging trees with different ranks:



Implementing Union-Find

- To “find” the cloud of vertex B, follow the parent pointer to vertex A, which is labeled as cloud 1.

```
// Finds the cloud of a given vertex
function find(x):
    while x != x.parent:
        x = x.parent
    return x
```



Implementing Union-Find

- How might you code `union(...)`? Here's how:

```
// Merges two clouds, given the root of each cloud
function union(root1, root2):
    if root1.rank > root2.rank:
        root2.parent = root1
    elif root1.rank < root2.rank:
        root1.parent = root2
    else:
        root2.parent = root1
        root1.rank++
```

Path Compression

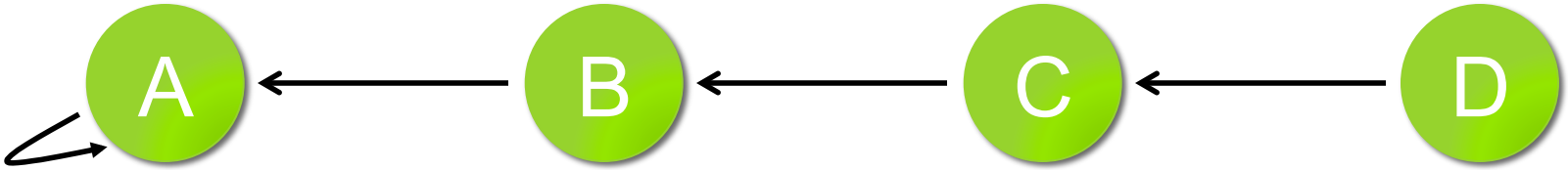
- The naive way of implementing Union-Find, which we just discussed, runs in $O(|E| \log |V|)$ time.

- We can bring this runtime down to amortized $O(1)$ by using path compression, a way of flattening the structure of the tree whenever `find(...)` is used on it.

Path Compression

- Say we have four vertices:

- We join these vertices into a tree such that we end up with the following structure:



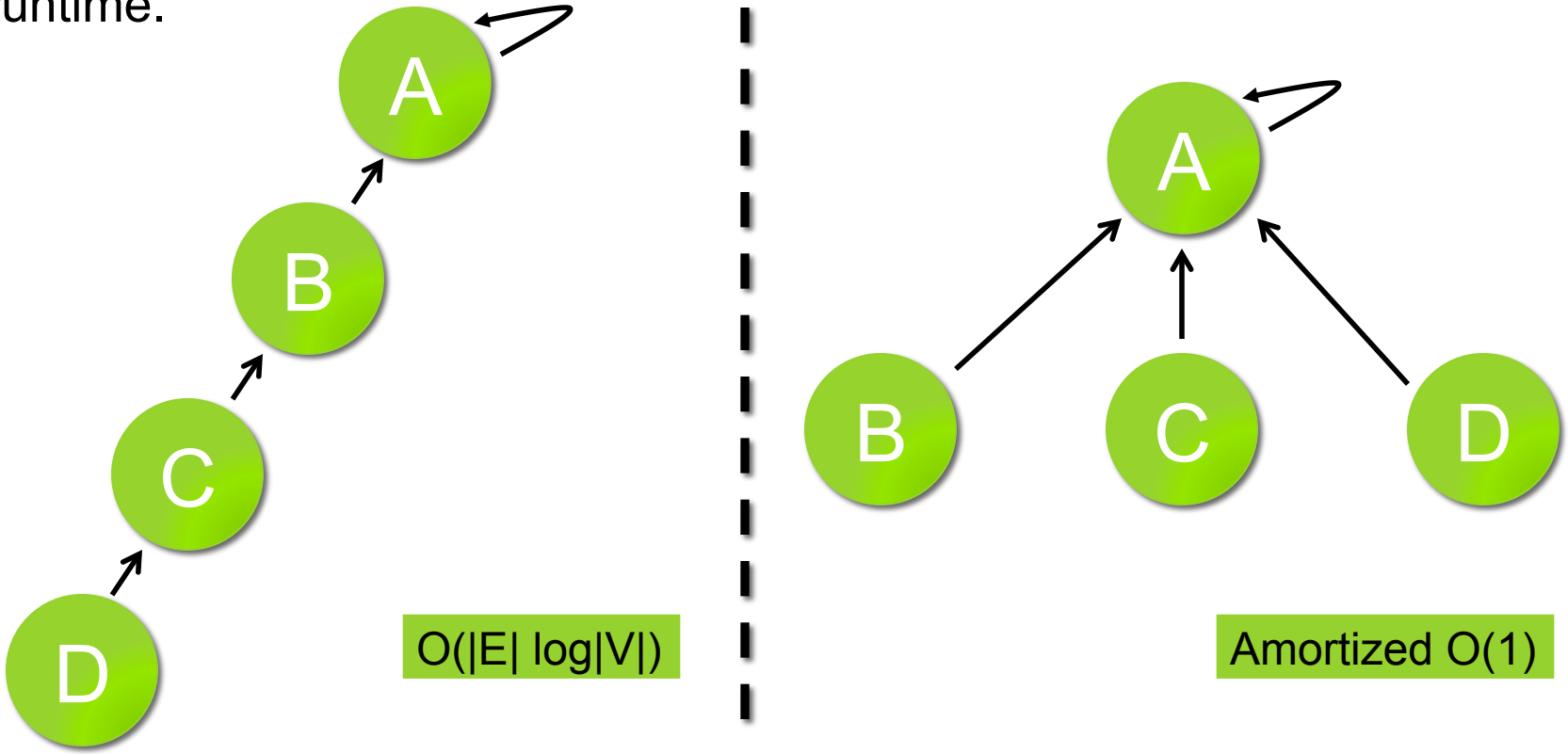
Path Compression

- Until now, whenever you wanted to lookup D's cloud, you'd have to follow a chain of pointers from D to A. It is this pointer chasing that slows down Union-Find.

- Notice that once a node ceases to be a root, it never resurfaces and its rank is fixed forever. Path compression exploits this key point to give us amortized $O(1)$ runtime.

Path Compression

- Instead of traversing up the tree every time D's cloud is asked for, we only do the work of searching for D's cloud once. As we follow the chain of parent pointers to A, we set the parent pointers of D and C to point to A. Next time we lookup D's cloud, we get guaranteed O(1) runtime.



Path Compression

- How might you code this nifty little trick? Here's how:

```
// Tweak find(...) to include path compression
function find(x):
    if x != x.parent:
        x.parent = find(x.parent)
    return x.parent
```

Runtime Analysis of Kruskal's with path compression

```
function kruskal(G):
 //Input: undirected, weighted graph G
 //Output: list of edges in MST
 for vertices v in G:
      makeCloud(v)          O(|V|)
 MST = []
 O(|E|log|E|) for sorting edges
 for all edges (u,v) in G sorted by weight: O(|E|)
    if u and v are not in same cloud:
        add (u,v) to MST
        merge clouds containing u and v amortized O(|1|)
 return MST
```

Runtime

- Total Runtime:
  - $O(|V|)$ for iterating through vertices
  - $O(|E| \log|E|)$ for sorting edges
  - $O(|E|) * O(|1|)$ for iterating through edges and merging clouds naively.
- $O(|V|) + O(|E| \log|E|) + O(|E|) * O(|1|)$

$$= O(|E| \log|E|)$$

- Total Runtime = $O(|E| \log|E|)$ – much better!

Proof of Correctness

- P(n): There exists an MST that contains the first n edges added by Kruskal's algorithm
- Base Case:
  - No edges have been added. P(0) is trivially true.
- Inductive Hypothesis:
  - Assume there exists an MST **M** that contains the first k edges added by Kruskal's algorithm, which form tree **T**
- Inductive Step:
  - Let **e** be the $(k+1)^{th}$ edge added to **T** by the algorithm.
  - If **e** is in **M**, **M** contains the first k+1 edges, so P(k+1) is true.
  - Otherwise, adding **e** to **M** must create a cycle. If this is the case, there must be another edge **e'** that is in this cycle that has not yet been added to **T**. Because the algorithm chose **e** instead of **e'**, we know that **e**.weight ≤ **e'**.weight. But because **e'** is part of **M** (an MST), we also know **e**.weight ≥ **e'**.weight.
  - This means **e**.weight = **e'**.weight, so we can replace **e'** with **e** in **M**, forming **M'**, which will still be minimal.
  - Since **M'** contains the first k+1 edges, P(k+1) is true.
- Because the base case holds true and P(k) → P(k+1), we have proved P(n) for all n ≥ 0 by induction.

Readings

- Dasgupta Section 5.1
  - Clear and concise explanations of MSTs and both algorithms discussed in this lecture