

LEC 18

CSE 373

# Minimum Spanning Trees

BEFORE WE START

---

Instructor

Hunter Schafer

TAs

Ken Aragon  
Khushi Chaudhari  
Joyce Elauria  
Santino Iannone  
Leona Kazi  
Nathan Lipiarski  
Sam Long  
Amanda Park


Paul Pham  
Mitchell Szeto  
Batina Shikhalieva  
Ryan Siu  
Elena Spasova  
Alex Teng  
Blarry Wang  
Aileen Zeng

# Learning Objectives

After this lecture, you should be able to...

1. Identify a Minimum Spanning Tree
2. Describe the Cycle and Cut properties of MSTs and explain how Prim's Algorithm utilizes the Cut property for its correctness.
3. Implement Prim's Algorithm and explain how it differs from Dijkstra's'

# Lecture Outline

- **Review Dijkstra's Algorithm, Topo Sort, Reductions** 
  - Correction to Topological Sort algorithm
- Minimum Spanning Trees
- Prim's Algorithm

# Implementing Dijkstra's: Pseudocode

- Use a MinPriorityQueue to keep track of the perimeter
  - Don't need to track entire graph
  - Don't need separate "known" set – implicit in PQ (we'll never try to update a "known" vertex)
- This pseudocode is much closer to what you'll implement in P4
  - However, still some details for you to figure out!
  - e.g. how to initialize distTo with all nodes mapped to  $\infty$
  - Spec will describe some optimizations for you to make 😊

```
dijkstraShortestPath(G graph, V start)
    Map edgeTo, distTo;
    initialize distTo with all nodes mapped to  $\infty$ , except start to 0

    PriorityQueue<V> perimeter; perimeter.add(start);

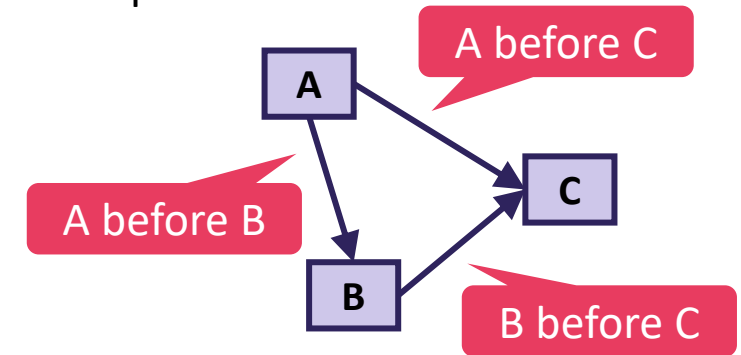
    while (!perimeter.isEmpty()):
        u = perimeter.removeMin()

        for each edge (u,v) to v with weight w:
            oldDist = distTo.get(v)           // previous best path to v
            newDist = distTo.get(u) + w       // what if we went through u?
            if (newDist < oldDist):
                distTo.put(v, newDist)
                edgeTo.put(v, u)
                if (perimeter.contains(v)):
                    perimeter.changePriority(v, newDist)
                else:
                    perimeter.add(v, newDist)
```

# (Review) Topological Sort

- A **topological sort** of a directed graph  $G$  is an ordering of the nodes, where for every edge in the graph, the origin appears before the destination in the ordering
- Intuition: a “dependency graph”
  - An edge  $(u, v)$  means  $u$  must happen before  $v$
  - A topological sort of a dependency graph gives an ordering that **respects dependencies**
- Applications:
  - Graduating
  - Compiling multiple Java files
  - Multi-job Workflows

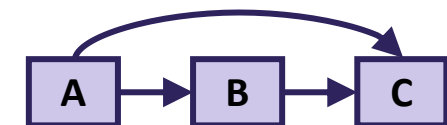
Input:



Topological Sort:



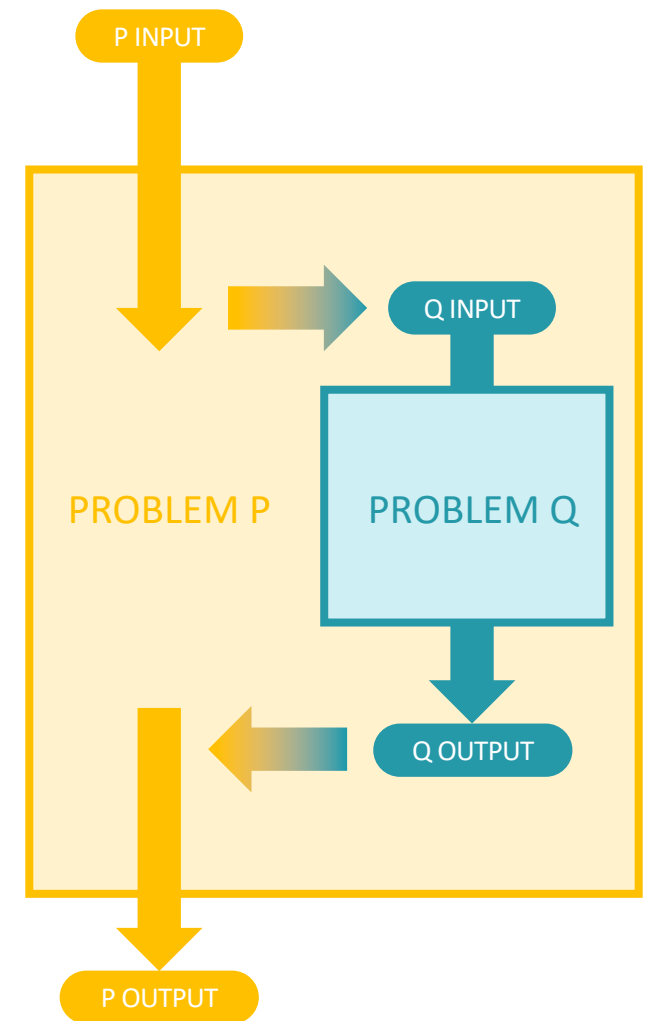
With original edges for reference:



# (Review) Reductions

- A **reduction** is a problem-solving strategy that involves using an algorithm for problem Q to solve a different problem P
  - Rather than modifying the algorithm for Q, we **modify the inputs/outputs** to make them compatible with Q!
  - “P reduces to Q”

- ➡ 1. Convert input for P into input for Q
- ↓ 2. Solve using algorithm for Q
- ← 3. Convert output from Q into output from P





# (Review) How To Perform Topo Sort?

- If we add a phantom “start” vertex pointing to other starts, we could use BFS!

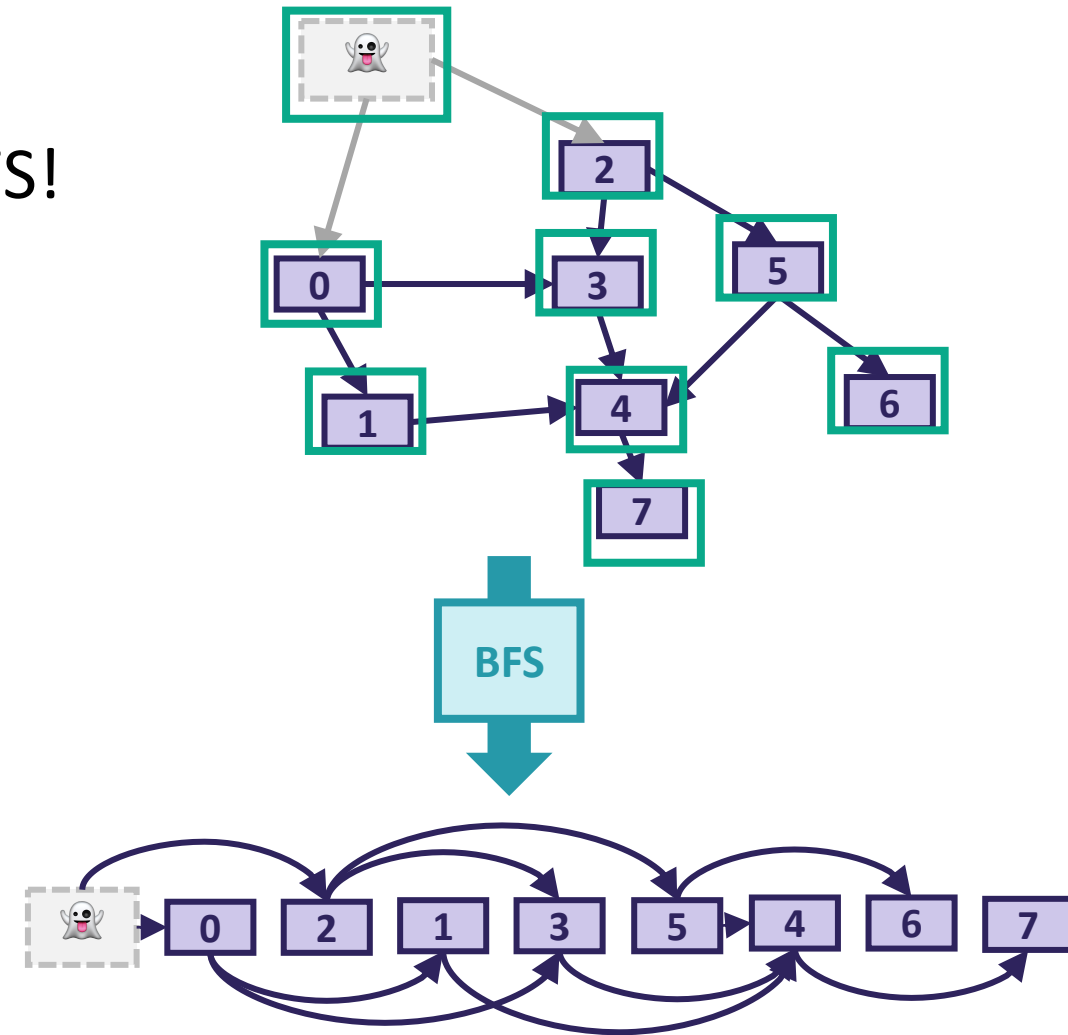
## IDEA 3

### Performing Topo Sort

Reduce topo sort to BFS by  
modifying graph, running BFS,  
then modifying output back

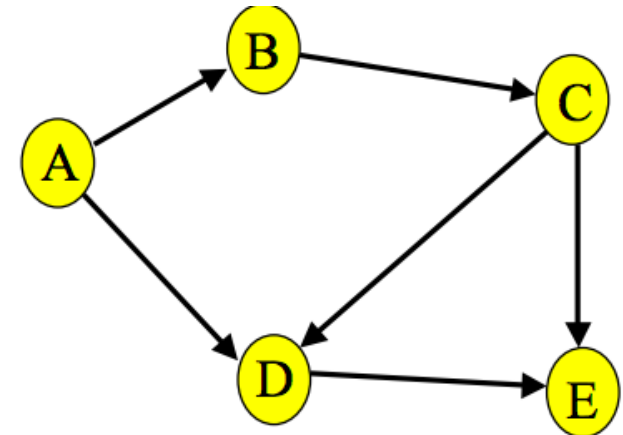
~~Swear~~ 

Also Wrong 



# Why BFS doesn't work here

- BFS always explores nodes level by level. Can visit a node that's close, even if there is a longer dependency chain to it ( $A \rightarrow B \rightarrow C \rightarrow D$ )
- How: Algorithm designers make mistakes too!
  - Showed up as a thought experiment in a previous offering's slides and then previous instructors (and I) picked it up assuming it would work!
- This is why computer scientists formally prove their algorithms work before publishing them!
  - Take CSE 417: Algorithms





# How to Topo Sort

We don't care for you to know these algorithms or why they work. We wanted to provide them since our last lecture taught you an incorrect algorithm!

## Khan's Algorithm

- Track in-degree of nodes
- Always explore nodes with in-degree 0
- When you mark a node as known, decrease the in-degree of all of its out-neighbors.

## Intuition

- Every (non-empty) DAG must have a node with in-degree 0.
- “Peel off” in-degree 0 nodes from the graph repeatedly to order by dependencies.

## Modified DFS

Run a simple DFS and add nodes to a visited list with two key details:

- Add the node to the beginning of the list
- Only add a node in a post-order fashion (after you have explore all of its children)

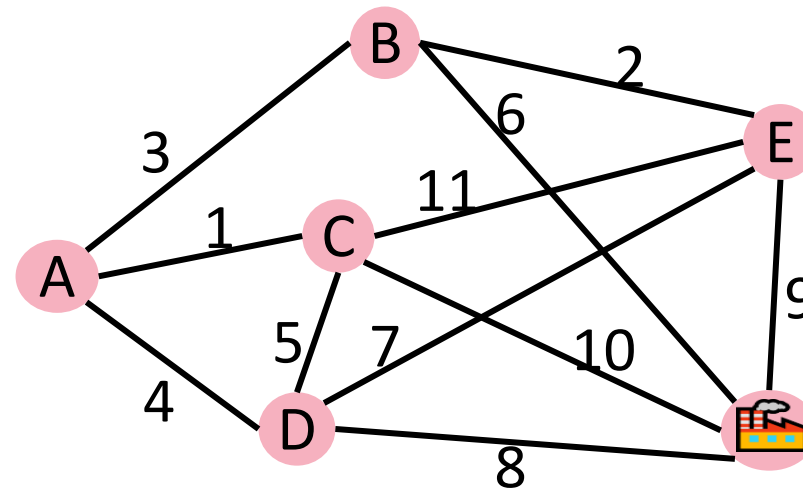
Run this multiple times until the whole graph has been visited, and you will magically have a topological sort!

# Lecture Outline

- *Review* Dijkstra's Algorithm, Topo Sort, Reductions
  - Correction to Topological Sort algorithm
- **Minimum Spanning Trees** 
- Prim's Algorithm

# Watt Would You Do? *(sorry, I know it hertz to read these puns)*

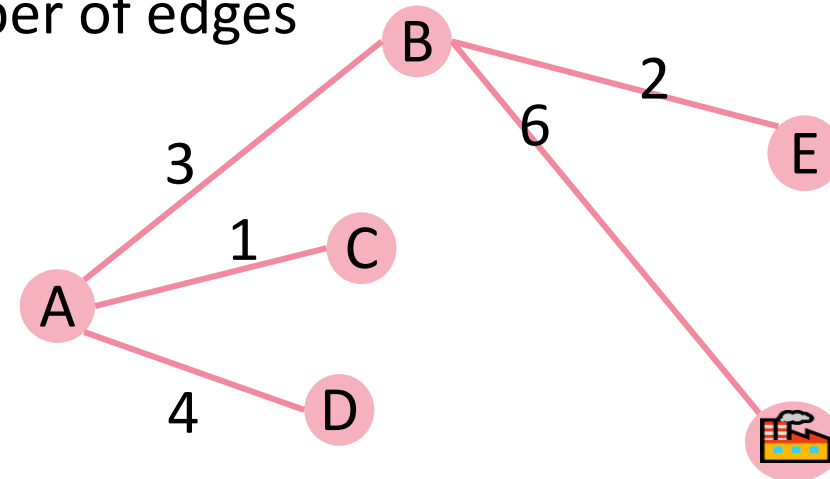
- Your friend at the electric company needs to connect all these cities to the power plant
- She knows the cost to lay wires between any pair of cities and wants the cheapest way to ensure electricity gets to every city



- Assume:
  - All edge weights are positive
  - The graph is undirected
  - Electricity can “travel through” cities

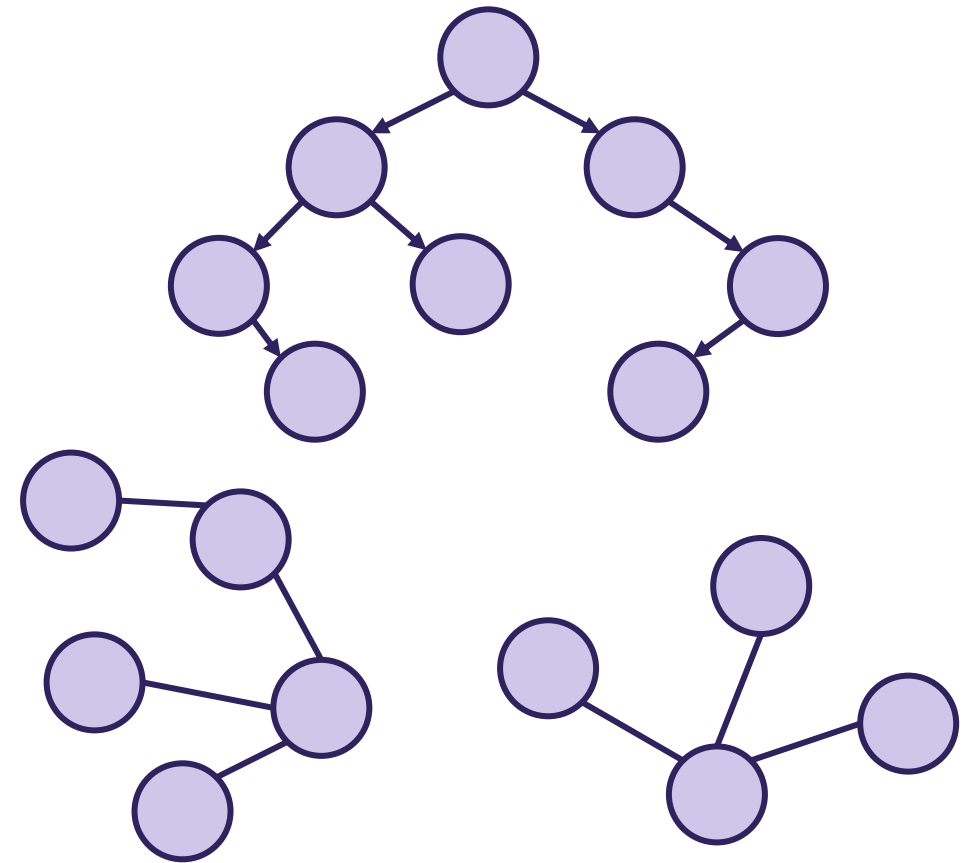
# Finding a Solution

- We need a *set of edges* such that:
  - Every vertex touches at least one edge (“the edges **span** the graph”)
  - The graph using just those edges is **connected**
  - The total weight of these edges is **minimized**
- Claim: The set of edges we pick never forms a cycle. Why?
  - $V-1$  edges is the exact minimum number of edges to connect all vertices
  - Taking away 1 edge breaks connectiveness
  - Adding 1 edge makes a cycle

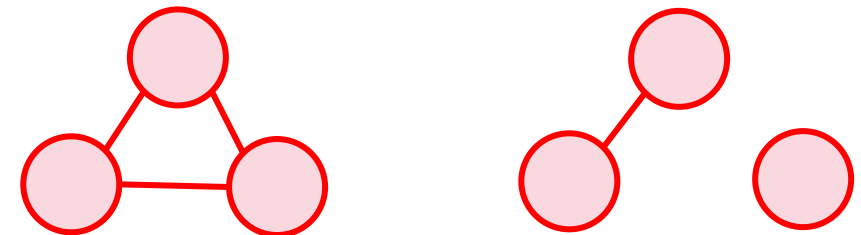


# Review Definition of a Tree

- So far, we've thought of trees as nodes with "parent" & "child" relationships
  - LEC 09: "A **binary tree** is a collection of nodes where each node has at most 1 parent and anywhere from 0 to 2 children"
- We can express the definition of a tree another way:
  - A **tree** is a collection of nodes connected by edges where there is exactly one path between any pair of nodes
  - So all trees are **connected**, **acyclic** graphs!



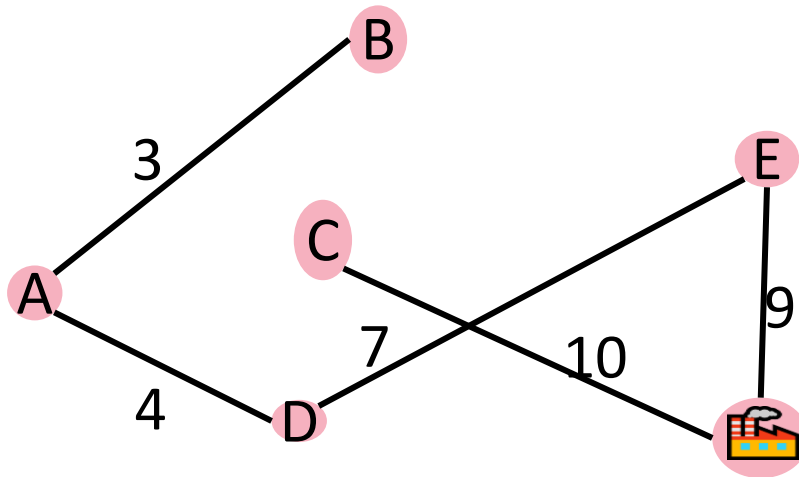
Not Trees:



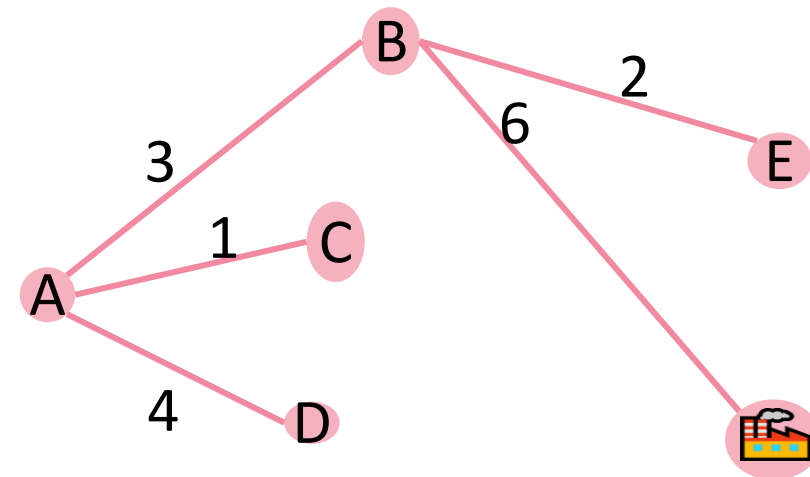
# Our Solution: The MST Problem

- We need a ~~set of edges such that~~ **Minimum Spanning Tree**:
  - Every vertex touches at least one edges (“the edges **span** the graph”)
  - The graph using just those edges is **connected**
  - The total weight of these edges is **minimized**

A Spanning Tree (cost 33):



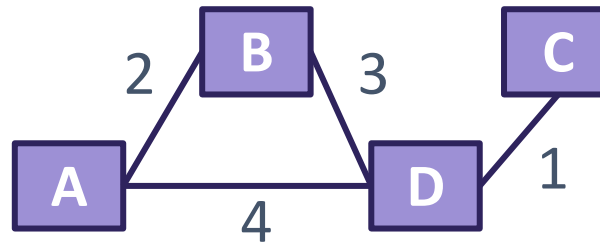
A Minimum Spanning Tree (cost 16):





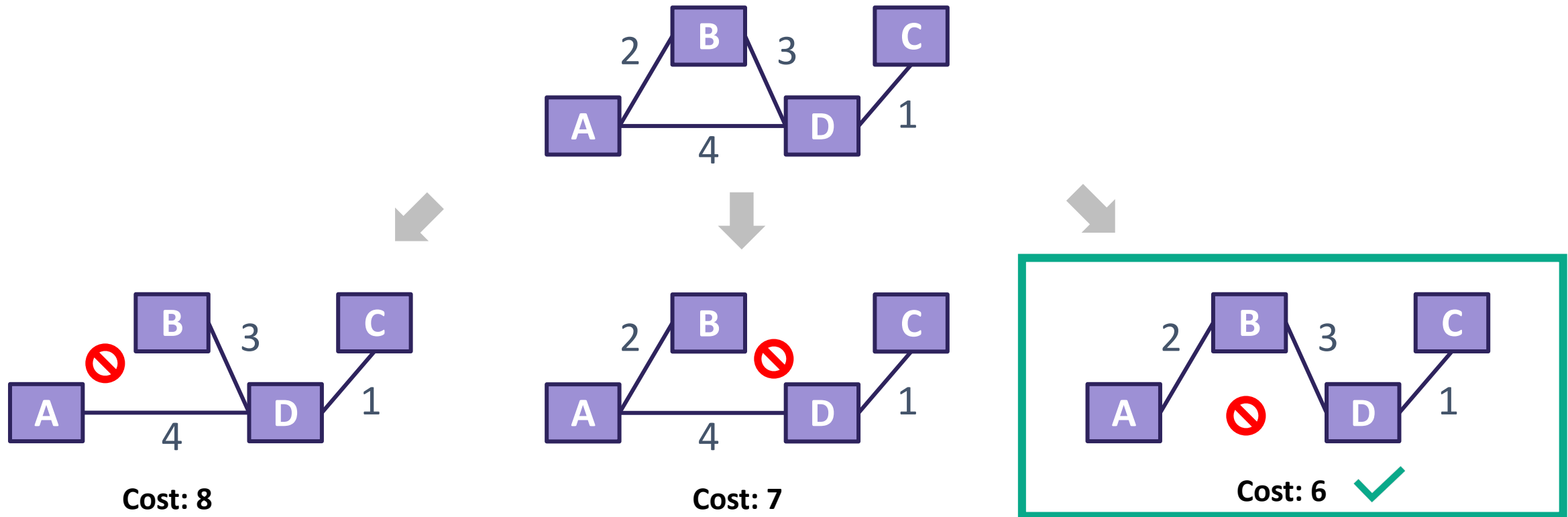
## You Try It!

What is the MST for the graph below? Enter its cost into ItemPool.



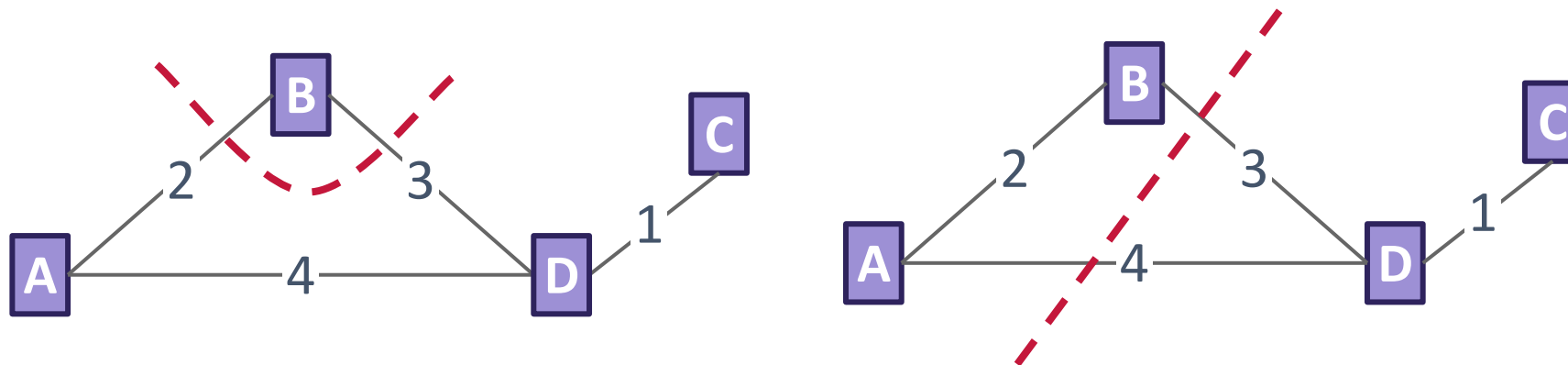
# Cycle Property

- Given any cycle, the heaviest edge along it must NOT be in the MST
  - Why not? A tree has no cycles, so we must discard at least one edge
  - Discarding exactly one edge will always leave all vertices connected
  - If we discard the heaviest edge, we minimize the edges still in use!



# Cut Property

- Given any **cut**, the minimum-weight crossing edge must be IN the MST
  - A **cut** is a partitioning of the vertices into two sets
  - (other crossing edges can also be in the MST)
  - Why? *Some* edge must connect the two, always best to use the smallest



🤔 *If only we knew of an algorithm that maintained a set of “known” and “unknown” vertices and repeatedly chose the minimum edge between the two sets ...*

# Lecture Outline

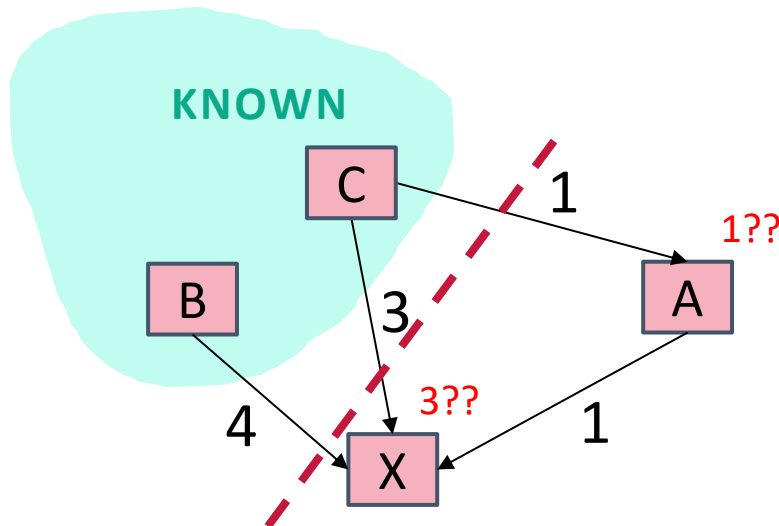
- Dijkstra's Algorithm
  - **Review** Definition & Examples
  - Implementing Dijkstra's
- Minimum Spanning Trees
- **Prim's Algorithm** 

# Adapting Dijkstra's: Prim's Algorithm

- MSTs don't have a "source vertex"
  - Replace "vertices for which we know the shortest path from  $s$ " with "vertices in the MST-under-construction"
  - Visit vertices in order of *distance from MST-under-construction*
  - Relax an edge based on its *distance from source*
- Note:
  - Prim's algorithm was developed in 1930 by Votěch Jarník, then independently rediscovered by Robert Prim in 1957 and Dijkstra in 1959. It's sometimes called Jarník's, Prim-Jarník, or DJP

# Adapting Dijkstra's: Prim's Algorithm

- Normally, Dijkstra's checks for a shorter path from the start.
- But MSTs don't care about individual paths, only the overall weight!
- New condition: "would this be a smaller edge to connect the current known set to the rest of the graph?"

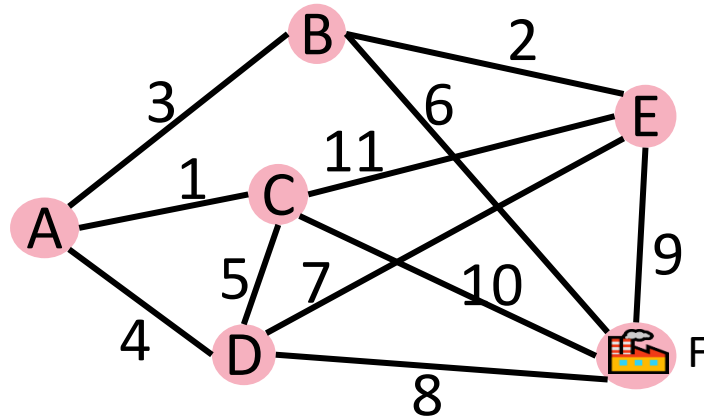


```
prims (G graph, V start)
Map edgeTo, distTo;
initialize distTo with all nodes mapped to  $\infty$ , except start to 0
PriorityQueue<V> perimeter; perimeter.add(start);

while (!perimeter.isEmpty()):
    u = perimeter.removeMin()
    known.add(u)
    for each edge (u,v) to unknown v with weight w:
        oldDist = distTo.get(v)           // previous smallest edge to v
        newDist = distTo.get(u) + w    // is this a smaller edge to v?
        if (newDist < oldDist):
            distTo.put(v, newDist)
            edgeTo.put(v, u)
            if (perimeter.contains(v)):
                perimeter.changePriority(v, newDist)
            else:
                perimeter.add(v, newDist)
```



# Let's Try It!



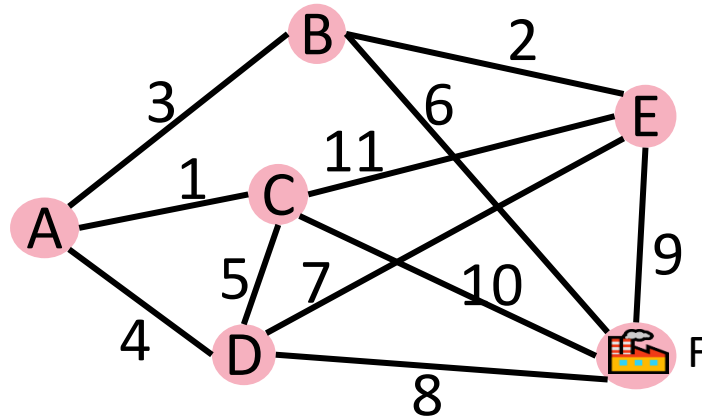
Node	known?	distTo	edgeTo
A			
B			
C			
D			
E			
F			

```
primMST(G graph, V start)
  Map edgeTo, distTo;
  initialize distTo to all  $\infty$ , except start to 0
  PriorityQueue<V> perimeter; add start;

  while (!perimeter.isEmpty()):
    u = perimeter.removeMin()
    known.add(u)
    for each edge (u,v) to unknown v with weight w:
      oldDist = distTo.get(v)
      newDist = w
      if (newDist < oldDist):
        distTo.put(v, newDist)
        edgeTo.put(v, u)
        if (perimeter.contains(v)):
          perimeter.changePriority(v, newDist)
        else:
          perimeter.add(v, newDist)
```

# Let's Try It!

Choose F as the start



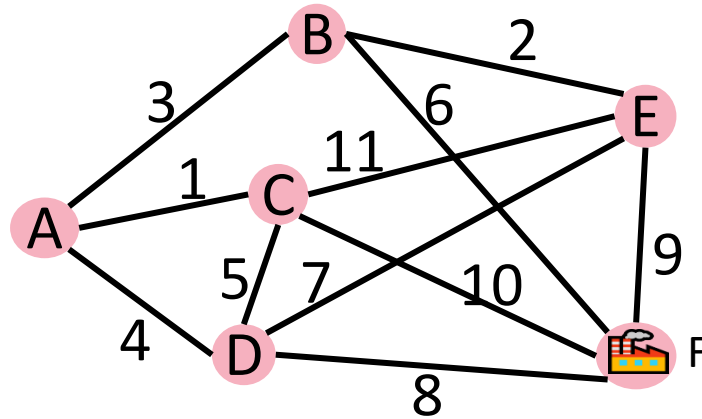
Node	known?	distTo	edgeTo
A		$\infty$	
B		$\infty$	
C		$\infty$	
D		$\infty$	
E		$\infty$	
F		0	/

```
primMST(G graph, V start)
  Map edgeTo, distTo;
  initialize distTo to all  $\infty$ , except start to 0
  PriorityQueue<V> perimeter; add start;

  while (!perimeter.isEmpty()):
    u = perimeter.removeMin()
    known.add(u)
    for each edge (u,v) to unknown v with weight w:
      oldDist = distTo.get(v)
      newDist = w
      if (newDist < oldDist):
        distTo.put(v, newDist)
        edgeTo.put(v, u)
        if (perimeter.contains(v)):
          perimeter.changePriority(v, newDist)
        else:
          perimeter.add(v, newDist)
```

# Let's Try It!

Pull F into the known set, updating its neighbors



Node	known?	distTo	edgeTo
A		$\infty$	
B		6??	F
C		10??	F
D		8??	F
E		9??	F
F	Y	0	/

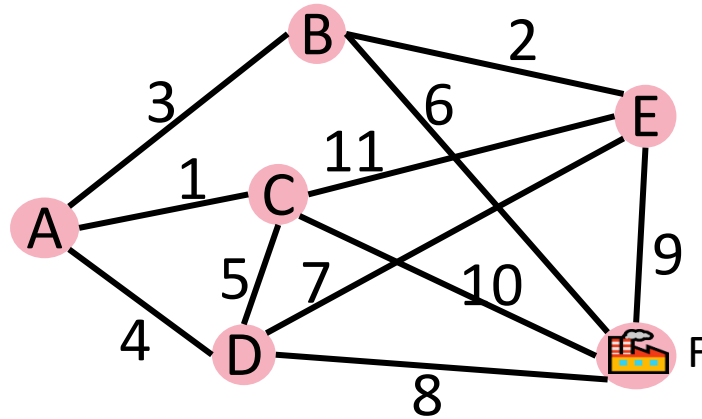
```

primMST(G graph, V start)
  Map edgeTo, distTo;
  initialize distTo to all  $\infty$ , except start to 0
  PriorityQueue<V> perimeter; add start;

  while (!perimeter.isEmpty()):
    u = perimeter.removeMin()
    known.add(u)
    for each edge (u,v) to unknown v with weight w:
      oldDist = distTo.get(v)
      newDist = w
      if (newDist < oldDist):
        distTo.put(v, newDist)
        edgeTo.put(v, u)
      if (perimeter.contains(v)):
        perimeter.changePriority(v, newDist)
      else:
        perimeter.add(v, newDist)
  
```

# Let's Try It!

Choose B and update its neighbors. Note that E is updated to 2, NOT 8 – only the cost to add it to the growing tree!



Node	known?	distTo	edgeTo
A		3??	B
B	Y	6	F
C		10??	F
D		8??	F
E		2??	B
F	Y	0	/

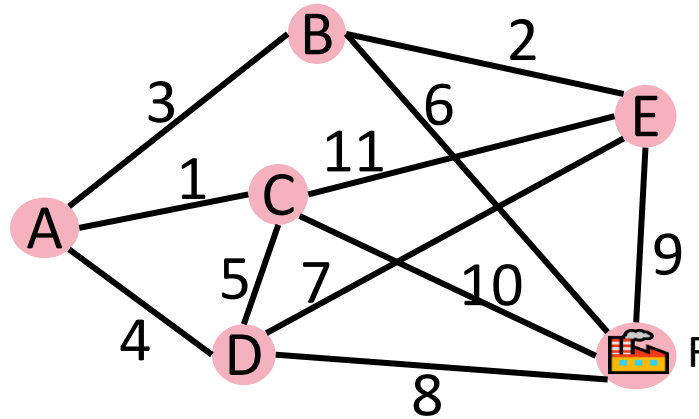
```

primMST(G graph, V start)
  Map edgeTo, distTo;
  initialize distTo to all  $\infty$ , except start to 0
  PriorityQueue<V> perimeter; add start;

  while (!perimeter.isEmpty()):
    u = perimeter.removeMin()
    known.add(u)
    for each edge (u,v) to unknown v with weight w:
      oldDist = distTo.get(v)
      newDist = w
      if (newDist < oldDist):
        distTo.put(v, newDist)
        edgeTo.put(v, u)
        if (perimeter.contains(v)):
          perimeter.changePriority(v, newDist)
        else:
          perimeter.add(v, newDist)
  
```

# Let's Try It!

Choose E and update its neighbors. We found a smaller way to get to D!



Node	known?	distTo	edgeTo
A		3??	B
B	Y	6	F
C		10??	F
D		7??	E
E	Y	2	B
F	Y	0	/

```

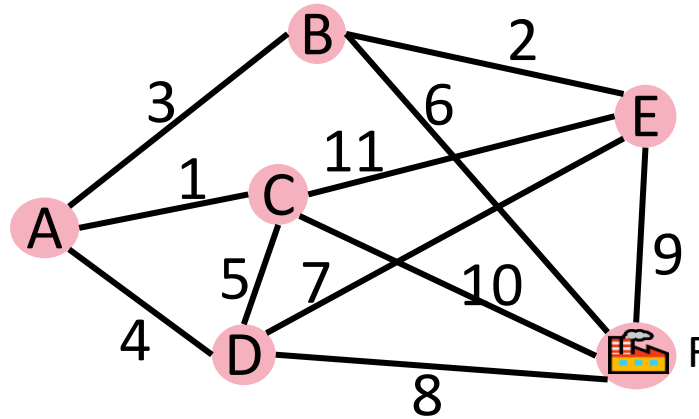
primMST(G graph, V start)
  Map edgeTo, distTo;
  initialize distTo to all  $\infty$ , except start to 0
  PriorityQueue<V> perimeter; add start;

  while (!perimeter.isEmpty()):
    u = perimeter.removeMin()
    known.add(u)
    for each edge (u,v) to unknown v with weight w:
      oldDist = distTo.get(v)
      newDist = w
      if (newDist < oldDist):
        distTo.put(v, newDist)
        edgeTo.put(v, u)
        if (perimeter.contains(v)):
          perimeter.changePriority(v, newDist)
        else:
          perimeter.add(v, newDist)

```

# Let's Try It!

Choose A and update its neighbors. We found much smaller options to add C and D!



Node	known?	distTo	edgeTo
A	Y	3	B
B	Y	6	F
C		1??	A
D		4??	A
E	Y	2	B
F	Y	0	/

```

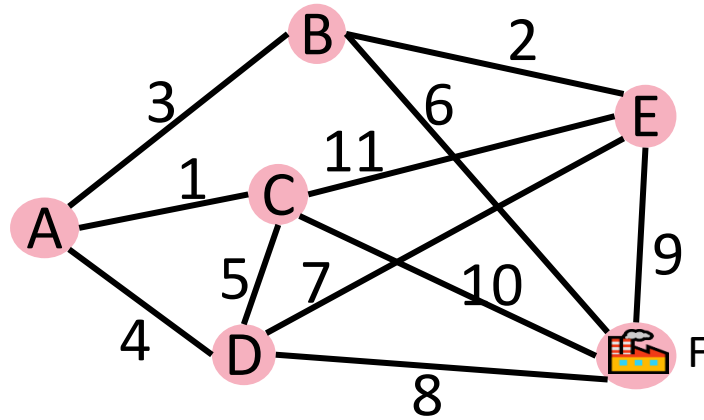
primMST(G graph, V start)
  Map edgeTo, distTo;
  initialize distTo to all  $\infty$ , except start to 0
  PriorityQueue<V> perimeter; add start;

  while (!perimeter.isEmpty()):
    u = perimeter.removeMin()
    known.add(u)
    for each edge (u,v) to unknown v with weight w:
      oldDist = distTo.get(v)
      newDist = w
      if (newDist < oldDist):
        distTo.put(v, newDist)
        edgeTo.put(v, u)
      if (perimeter.contains(v)):
        perimeter.changePriority(v, newDist)
      else:
        perimeter.add(v, newDist)
  
```



# Let's Try It!

Choose C and  
update its  
neighbors.  
Nothing  
changes.



Node	known?	distTo	edgeTo
A	Y	3	B
B	Y	6	F
C	Y	1	A
D		4??	A
E	Y	2	B
F	Y	0	/

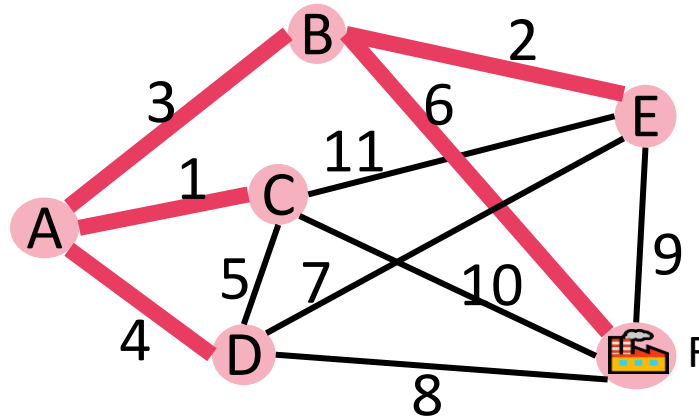
```

primMST(G graph, V start)
  Map edgeTo, distTo;
  initialize distTo to all  $\infty$ , except start to 0
  PriorityQueue<V> perimeter; add start;

  while (!perimeter.isEmpty()):
    u = perimeter.removeMin()
    known.add(u)
    for each edge (u,v) to unknown v with weight w:
      oldDist = distTo.get(v)
      newDist = w
      if (newDist < oldDist):
        distTo.put(v, newDist)
        edgeTo.put(v, u)
        if (perimeter.contains(v)):
          perimeter.changePriority(v, newDist)
        else:
          perimeter.add(v, newDist)
  
```

# Let's Try It!

Choose D and finish the algorithm! We have our MST: an undirected graph with all edgeTo edges!



Node	known?	distTo	edgeTo
A	Y	3	B
B	Y	6	F
C	Y	1	A
D	Y	4	A
E	Y	2	B
F	Y	0	/

```

primMST(G graph, V start)
  Map edgeTo, distTo;
  initialize distTo to all  $\infty$ , except start to 0
  PriorityQueue<V> perimeter; add start;

  while (!perimeter.isEmpty()):
    u = perimeter.removeMin()
    known.add(u)
    for each edge (u,v) to unknown v with weight w:
      oldDist = distTo.get(v)
      newDist = w
      if (newDist < oldDist):
        distTo.put(v, newDist)
        edgeTo.put(v, u)
        if (perimeter.contains(v)):
          perimeter.changePriority(v, newDist)
        else:
          perimeter.add(v, newDist)
  
```

# Prim's Runtime

Final result:

$$\Theta(|V| \log |V| + |E| \log |V|)$$

Unsurprisingly, runtime is just like Dijkstra's algorithm.

$$\Theta(|V| \log |V|)$$

$\Theta(|V|)$  iterations

$$\Theta(\log |V|)$$

total  $\Theta(|E|)$  iterations

$$\Theta(1)$$

$$\Theta(\log |V|)$$

$$\Theta(\log |V|)$$

$$\Theta(|E| \log |V|)$$

```
primMST(G graph, V start)
```

```
    Map edgeTo, distTo;
```

$\Theta(|V|)$  → initialize distTo with all nodes mapped to infinity

```
    PriorityQueue<V> perimeter; perimeter.add(start);
```

```
    while (!perimeter.isEmpty()):
```

```
        u = perimeter.removeMin()
```

```
        known.add(u)
```

```
        for each edge (u,v) to unknown v with weight w:
```

```
            oldDist = distTo.get(v)
```

```
            newDist = w
```

```
            if (newDist < oldDist):
```

```
                distTo.put(v, newDist)
```

```
                edgeTo.put(v, u)
```

```
            if (perimeter.contains(v)):
```

```
                perimeter.changePriority(v, newDist)
```

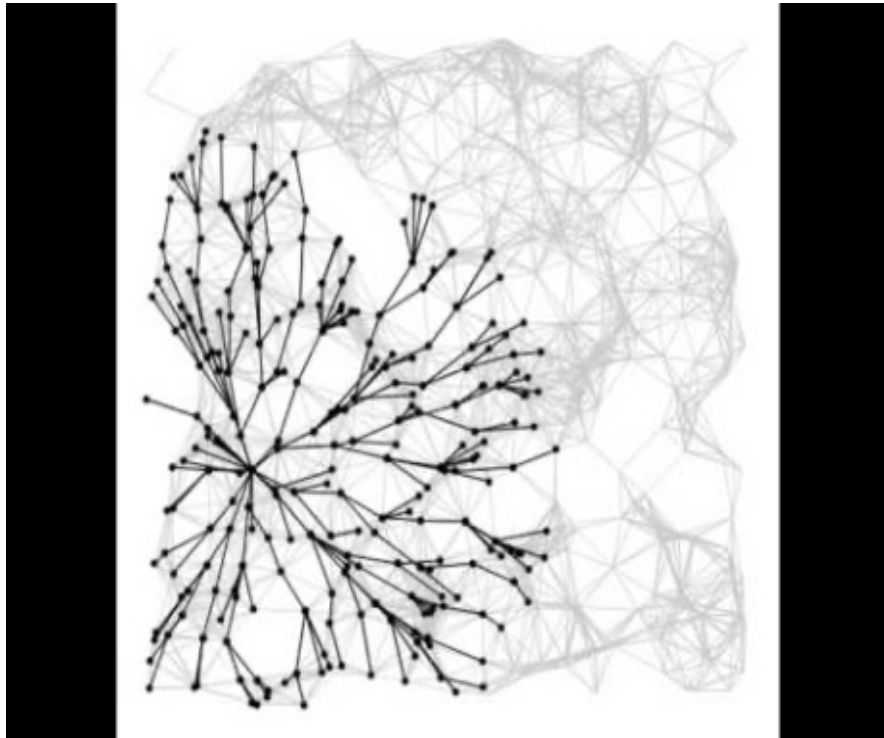
```
            else:
```

```
                perimeter.add(v, newDist)
```

# Prim's Demos and Visualizations

## Dijkstra's Algorithm

Dijkstra's proceeds radially from its source, because it chooses edges by *path length from source*



## Prim's Algorithm

Prim's jumps around the graph (the perimeter), because it chooses edges by *edge weight* (there's no source)

