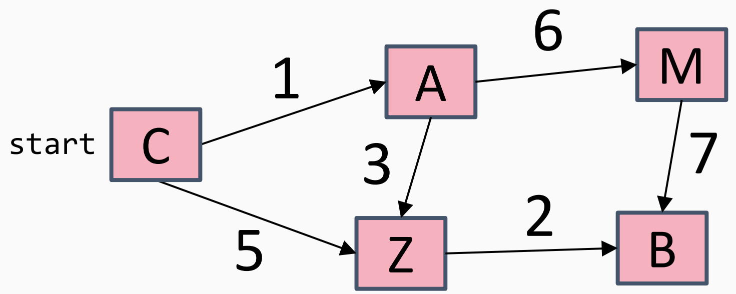
LEC 17

CSE 373

Minimum Spanning Trees

BEFORE WE START

What order will we visit these vertices while running Dijkstra's on this graph?



pollev.com/uwcse373

Instructor Aaron Johnston

TAs Timothy Akintilo, Brian Chan, Joyce Elauria, Eric Fan, Farrell Fileas, Melissa Hovik, Leona Kazi, Keanu Vestil, Siddharth Vaidyanathan, Howard Xiao

Announcements

- P3 due this Wednesday, 8/05 11:59pm PDT
- EX3 due this Friday, 8/07 11:59pm PDT
- P4 Partner Form is now open! Please fill out by lecture on Wednesday

Announcements: Exam I

- Exam I feedback & solution will be published this evening
  - We were so impressed with the skills everyone demonstrated! It's wonderful to see how much everyone has learned this quarter ☺
  - Mean: ~85%, Median: ~87%, Standard Deviation: ~10%
  - Remember, this exam is just <u>one piece of feedback</u> about your learning this quarter (e.g. didn't test you on writing code at all, but that's clearly an important set of learning objectives from this course!)
  - Grades in general are still a woefully incomplete signal for you to gauge your mastery of the learning objectives, and <u>do not whatsoever indicate</u> some kind of fictional "computer science ability".
  - Use this as an opportunity to get feedback and review what you got wrong to further your learning! The grade you get is <u>so much less important</u> than what you do with it afterward!
  - Regrade requests will open tomorrow evening. Make sure you review the sample solution first.

Learning Objectives

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

1. Describe the runtime for Dijkstra's algorithm and explain where it comes from
2. Identify a Minimum Spanning Tree, and explain why the Cut and Cycle properties must be true from the definition of an MST
3. Implement Prim's Algorithm and explain how it differs from Dijkstra's
4. Describe Kruskal's Algorithm at a high level, explain why it works, and describe why it needs a new ADT

Lecture Outline

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

Review Our Graph Problem Collection

WED

s-t Connectivity Problem

Given source vertex **s** and a target vertex **t**, does there exist a path from **s** to **t**?

SOLUTION
Base Traversal: BFS or DFS
Modification: Check if each vertex == t

WED

Unweighted Shortest Path Problem

Given source vertex **s** and target vertex **t**, what path from s to t **minimizes the number of edges**? How long is that path, and what edges make it up?

SOLUTION
Base Traversal: BFS
Modification: Generate shortest path tree as we go

FRI

Weighted Shortest Path Problem

Given source vertex **s** and target vertex **t**, what path from s to t **minimizes the total weight of its edges**? How long is that path, and what edges make it up?

SOLUTION
Base Traversal: Dijkstra's Algorithm
Modification: Generate shortest path tree as we go

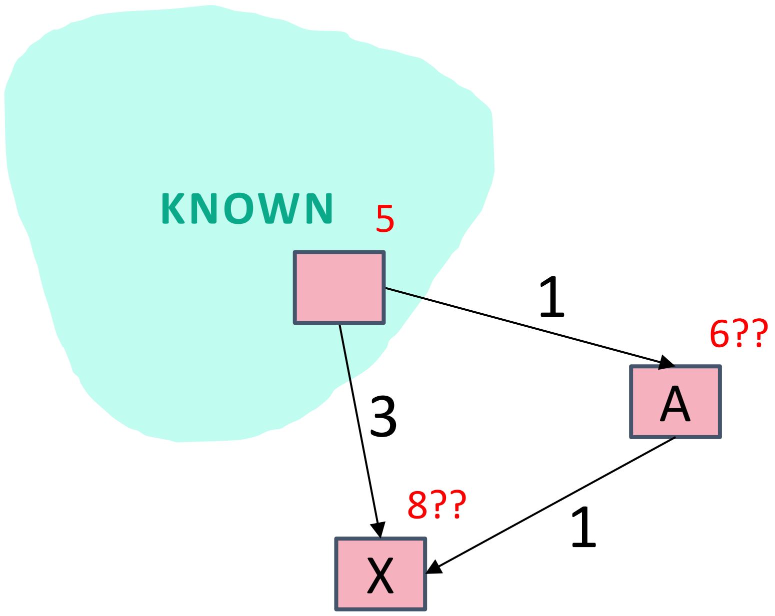
Review Dijkstra's Algorithm: Key Properties

- Once a vertex is marked known, its shortest path is known
  - Can reconstruct path by following back-pointers (in edgeTo map)
- While a vertex is not known, another shorter path might be found
  - We call this update **relaxing** the distance because it only ever shortens the current best path
- Going through closest vertices first lets us confidently say no shorter path will be found once known
  - Because not possible to find a shorter path that uses a farther vertex we'll consider later

```
dijkstraShortestPath(G graph, V start)
  Set known; Map edgeTo, distTo;
  initialize distTo with all nodes mapped to ∞, except start to 0

  while (there are unknown vertices):
    let u be the closest unknown vertex
    known.add(u)
    for each edge (u,v) to unknown 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)
```

CORRECTED AFTER LECTURE

Review Why Does Dijkstra's Work?



INVARIANT

Dijkstra's Algorithm Invariant

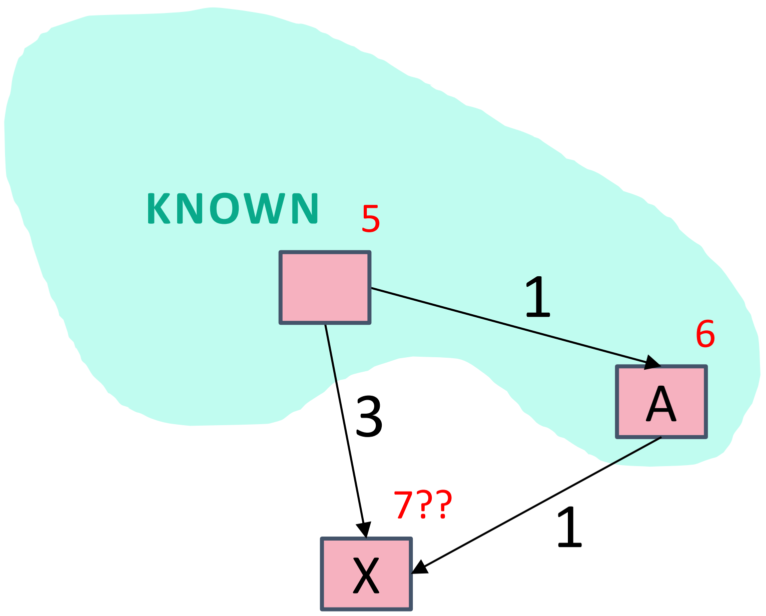
All vertices in the "known" set have the correct shortest path

- Similar "First Try Phenomenon" to BFS
- How can we be sure we won't find a shorter path to X later?

Example:

- We're about to add X to the known set
- But how can we be sure we won't later find a path through some node A that is shorter to X?

Review Why Does Dijkstra's Work?



Example:

- We're about to add X to the known set
- But how can we be sure we won't later find a path through some node A that is shorter to X?
- Because *if we could, Dijkstra's would explore A first*

INVARIANT

Dijkstra's Algorithm Invariant

All vertices in the "known" set have the correct shortest path

- Similar "First Try Phenomenon" to BFS
- How can we be sure we won't find a shorter path to X later?
  - **Key Intuition**: Dijkstra's works because:
  - IF we always add the closest vertices to "known" first,
  - THEN by the time a vertex is added, any possible relaxing has happened and the path we know is *always the shortest*!

Lecture Outline

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

Implementing Dijkstra's

- How do we implement "let u be the closest unknown vertex"?

- Would sure be convenient to store vertices in a structure that…
  - Gives them each a distance "priority" value
  - Makes it fast to grab the one with the smallest distance
  - Lets us update that distance as we discover new, better paths

MIN PRIORITY QUEUE ADT

```
dijkstraShortestPath(G graph, V start)
  Set known; Map edgeTo, distTo;
  initialize distTo with all nodes mapped to ∞, except start to 0

  while (there are unknown vertices):
    let u be the closest unknown vertex
    known.add(u)
    for each edge (u,v) to unknown 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(u, newDist)
        edgeTo.put(u, v)
```

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 ∞
  - Spec will describe some optimizations for you to make ☺

```
dijkstraShortestPath(G graph, V start)
  Map edgeTo, distTo;
  initialize distTo with all nodes mapped to ∞, 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)
```

CORRECTED AFTER LECTURE

Dijkstra's Runtime

```
dijkstraShortestPath(G graph, V start)
  Map edgeTo, distTo;
  initialize distTo with all nodes mapped to ∞, 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)
```

$\Theta(|V|)$ → initialize distTo with all nodes mapped to ∞, except start to 0

$\Theta(|V|\log|V|)$ $\begin{cases} \Theta(|V|) \text{ iterations} \to \text{while} \\ \Theta(\log|V|) \to \text{u = perimeter.removeMin()} \end{cases}$

$\Theta(|E|\log|V|)$ $\begin{cases} \underline{\text{total}}\ \Theta(|E|) \text{ iterations} \to \text{for each edge} \\ \Theta(1) \to \text{oldDist} \ldots \text{edgeTo.put(v, u)} \\ \Theta(\log|V|) \to \text{perimeter.changePriority(v, newDist)} \\ \Theta(\log|V|) \to \text{perimeter.add(v, newDist)} \end{cases}$

Dijkstra's Runtime

Final result:

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

Why can't we simplify further?

- We don't know if |V| or |E| is going to be larger, so we don't know which term will dominate.
- *Sometimes we assume |E| is larger than |V|, so |E|log|V| dominates. But not always true!*

$\Theta(|V|\log|V|)$ — $\Theta(|V|)$ iterations, $\Theta(\log|V|)$

$\Theta(|E|\log|V|)$ — total $\Theta(|E|)$ iterations, $\Theta(1)$, $\Theta(\log|V|)$, $\Theta(\log|V|)$

$\Theta(|V|)$

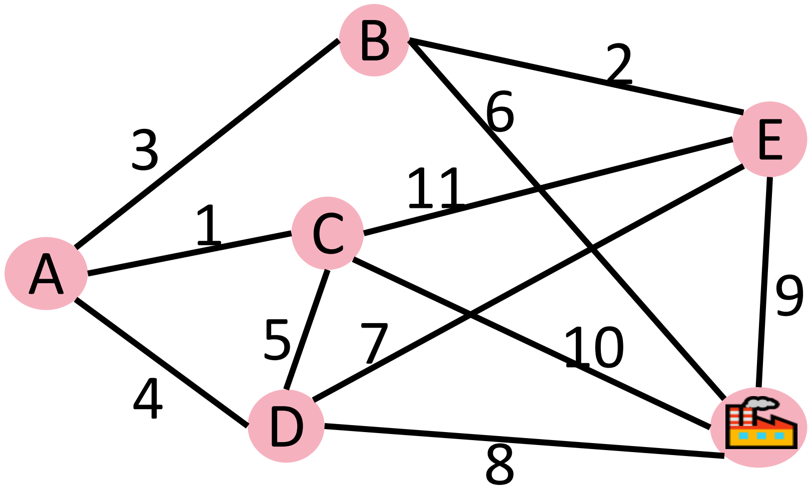
```
dijkstraShortestPath(G graph, V start)
  Map edgeTo, distTo;
  initialize distTo with all nodes mappe

  PriorityQueue<V> perimeter; perimeter.

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

    for each edge (u,v) to v with weight
      oldDist = distTo.get(v)      // pr
      newDist = distTo.get(u) + w  // wh
      if (newDist < oldDist):
        distTo.put(v, newDist)
        edgeTo.put(v, u)
        if (perimeter.contains(v)):
          perimeter.changePriority(v, ne
        else:
          perimeter.add(v, newDist)
```

Lecture Outline

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

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

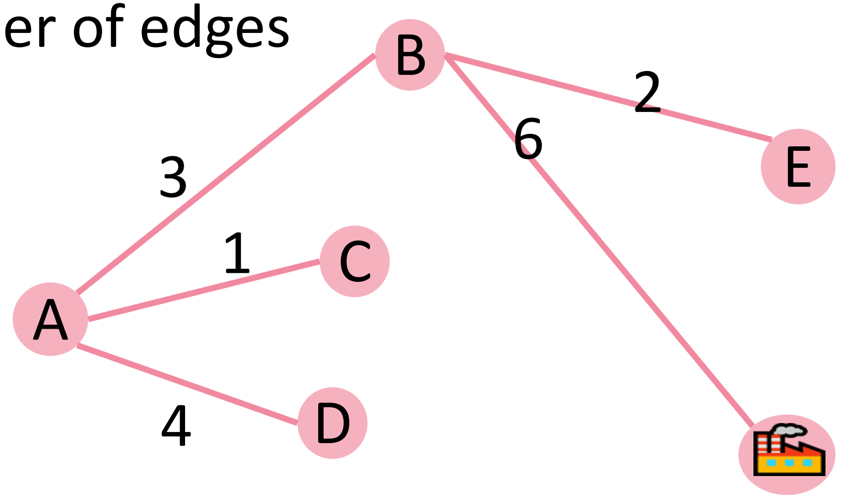


- Assume:
  - All edge weights are positive
  - The graph is undirected

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

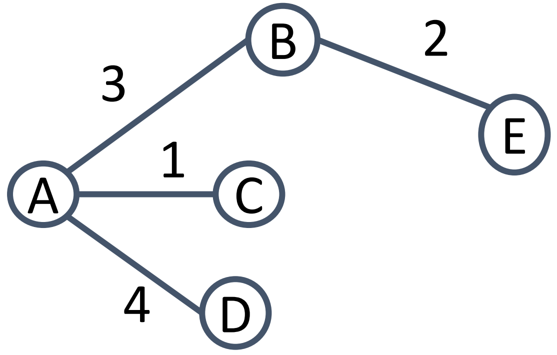


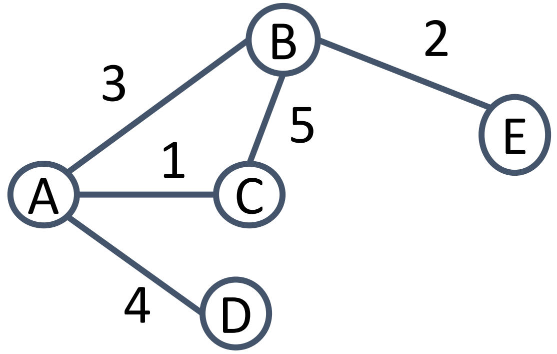
Poll Everywhere

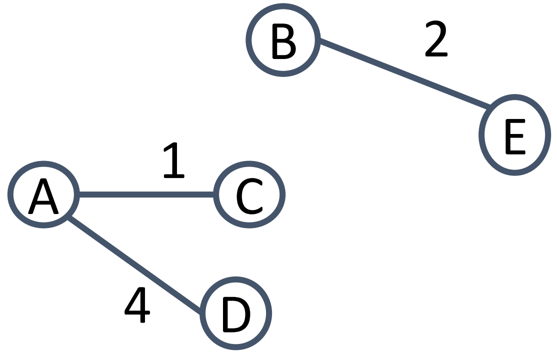
pollev.com/uwcse373

Which of these are trees?

A. Tree / Not-Tree / Not-Tree
B. Tree / Tree / Not-Tree
C. Tree / Not-Tree / Tree
D. Tree / Tree / Tree
E. I’m not sure …

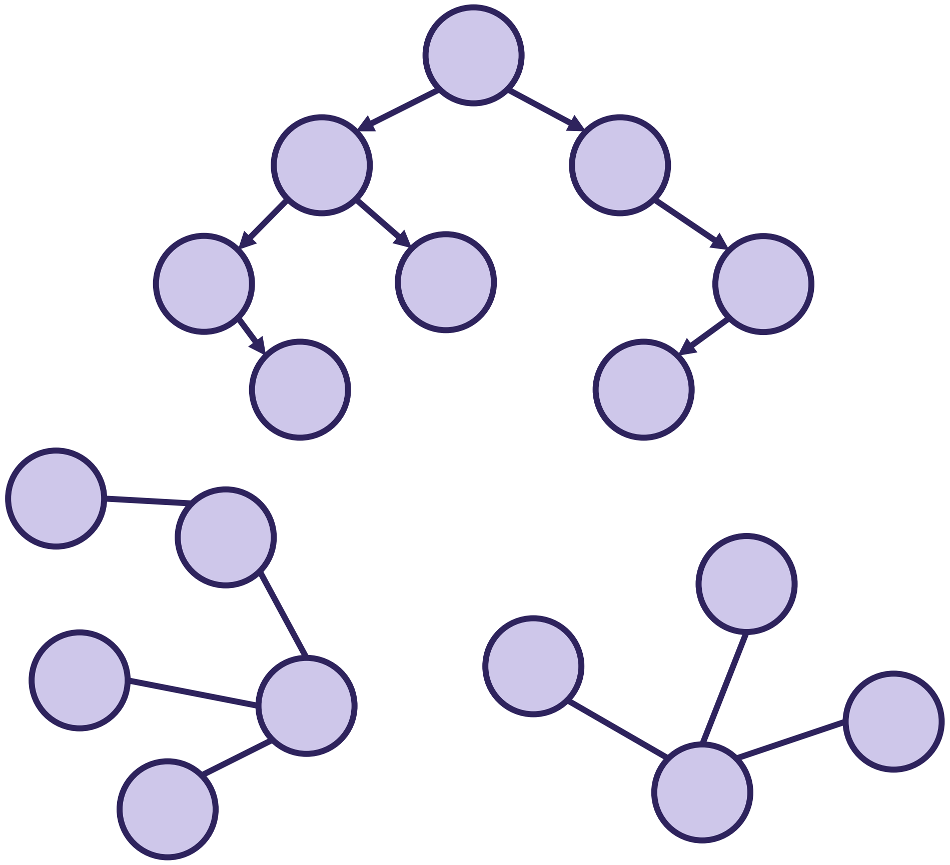




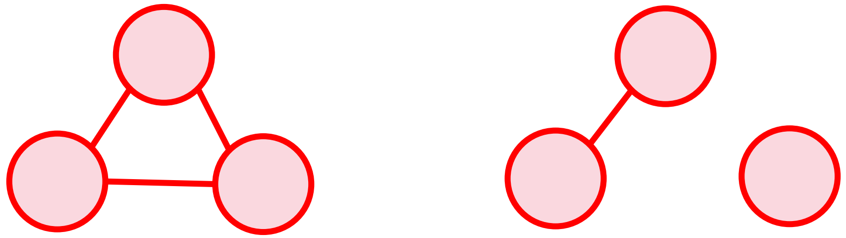


Review Definition of a Tree

- 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!



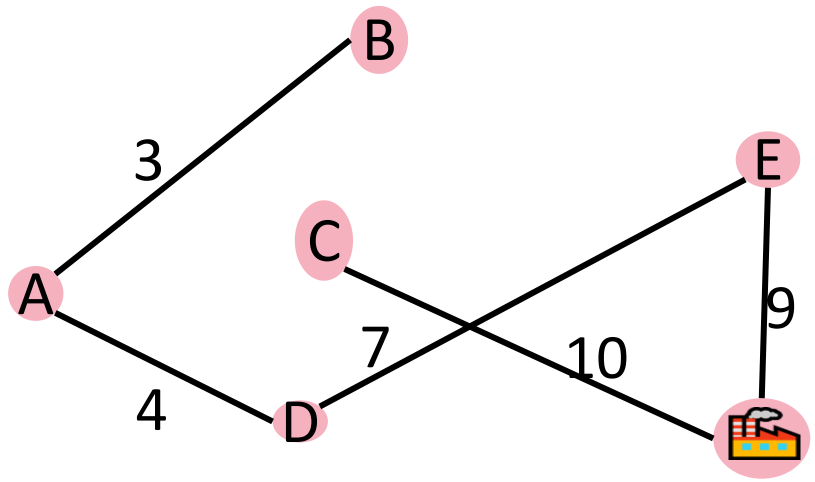
Not Trees:



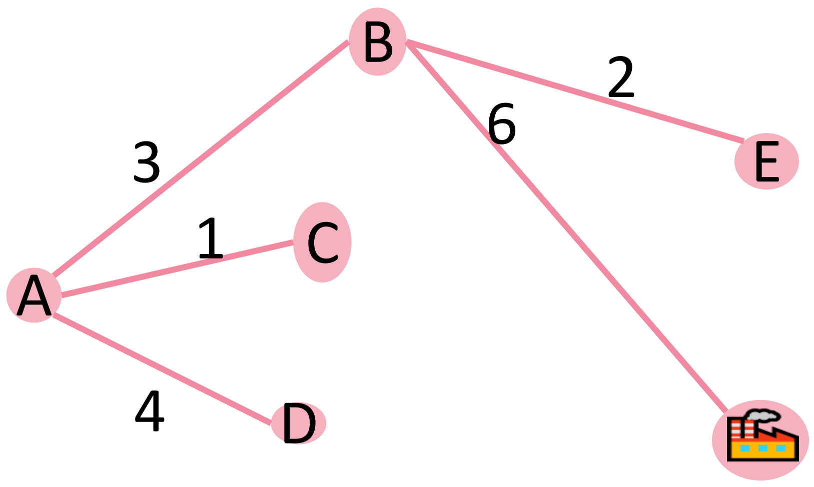
Our Solution: The MST Problem

- 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:

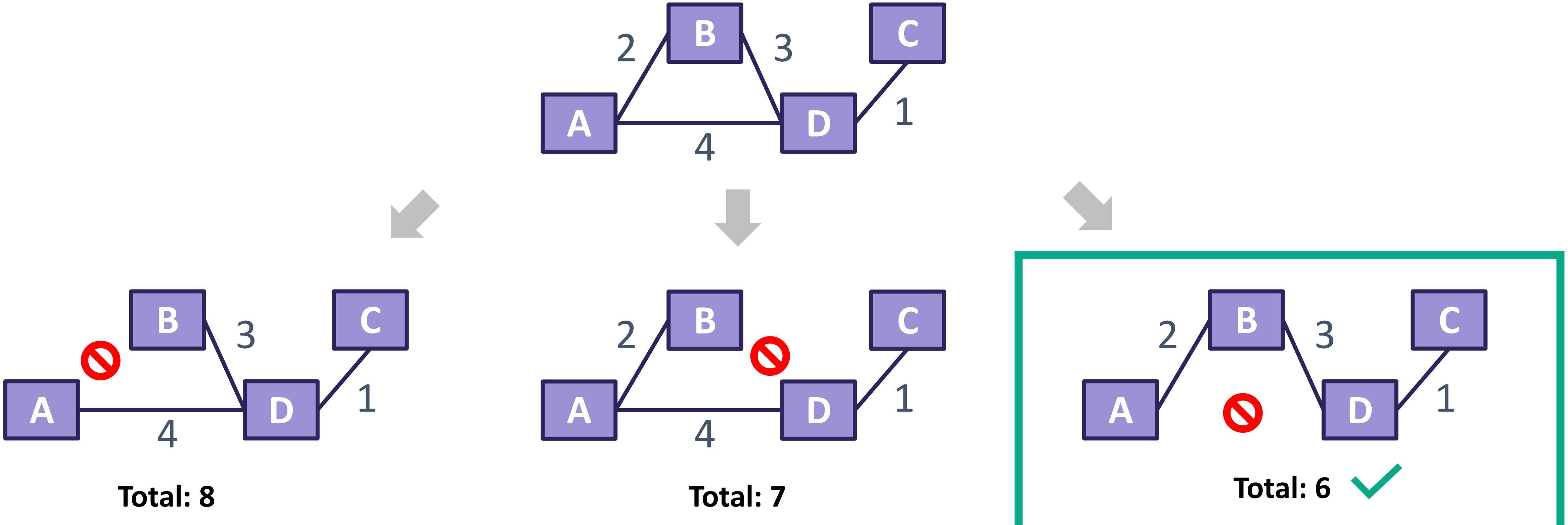


A Minimum Spanning Tree:



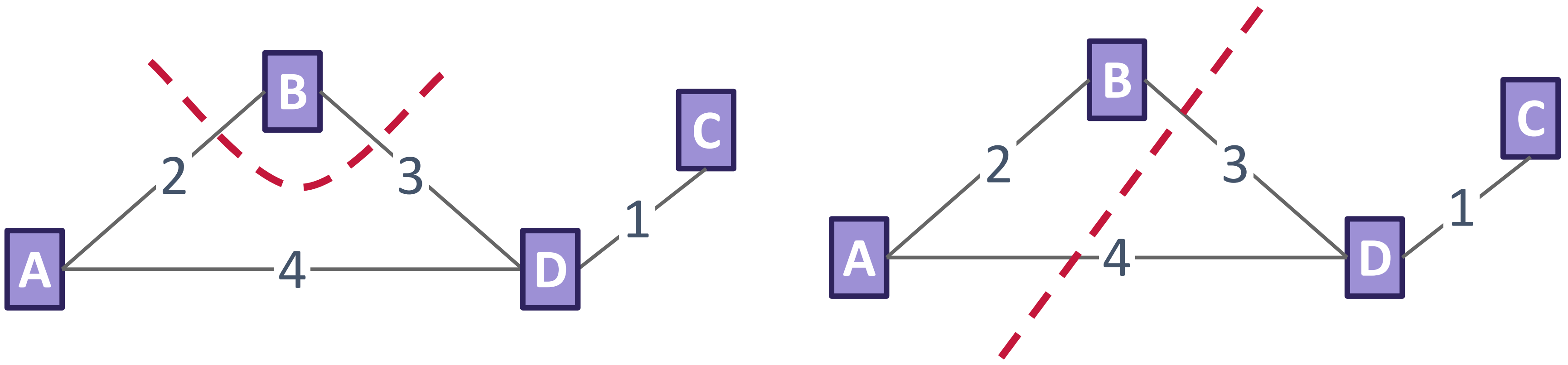
Cycle Property

- 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**
- Kruskal's Algorithm & Disjoint Sets

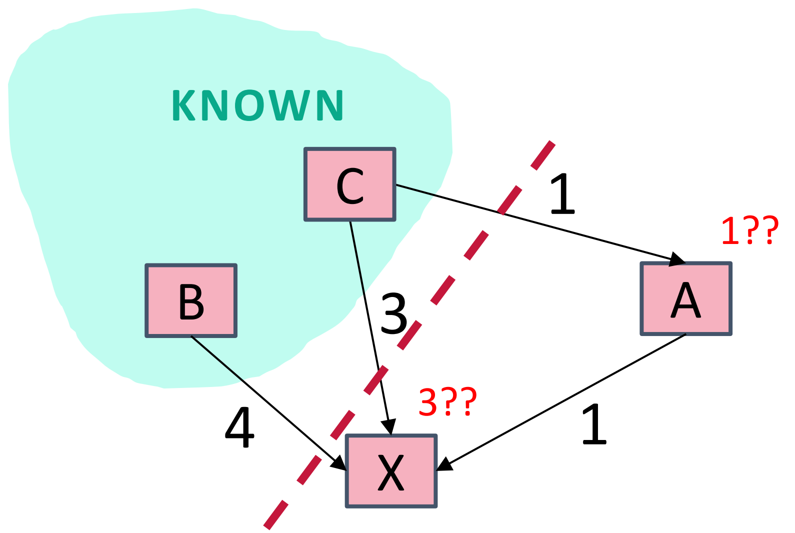
Adapting Dijkstra's: Prim's Algorithm

- 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?"

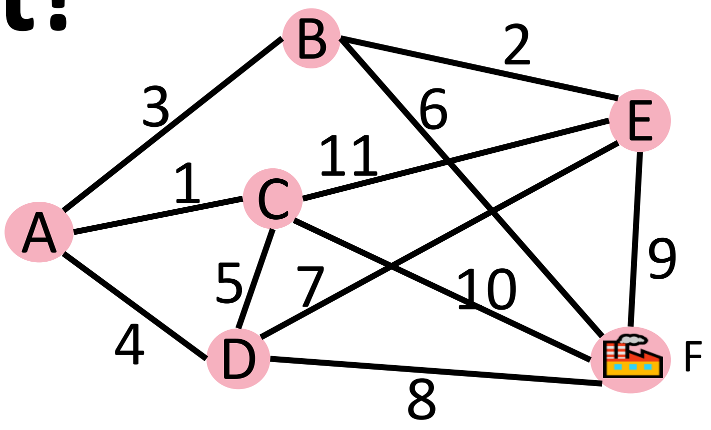



```
prims (G graph, V start)
Map edgeTo, distTo;
initialize distTo with all nodes mapped to ∞, except start to 0
PriorityQueue<V> perimeter; perimeter.add(start);

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)
```

Poll Everywhere

pollev.com/uwcse373

Let’s Try It!



Node	known?	distTo	edgeTo
A			
B			
C			
D			
E			
F			

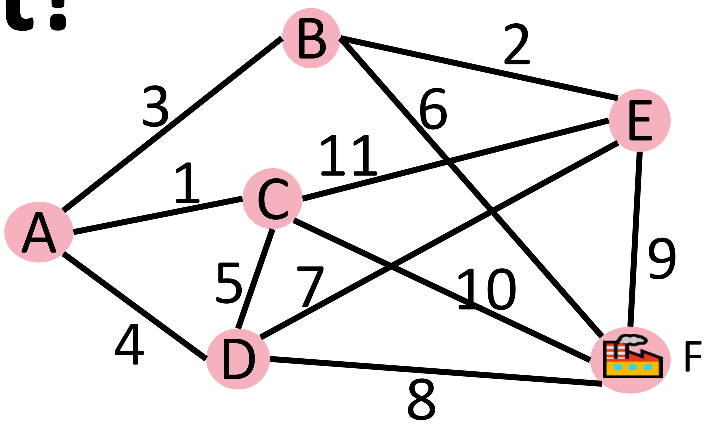
```
primMST(G graph, V start)
  Map edgeTo, distTo;
  initialize distTo to all ∞, except start to 0
  PriorityQueue<V> perimeter; add start;

  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)
```

Poll Everywhere

pollev.com/uwcse373

Let's Try It!

Choose F as the start

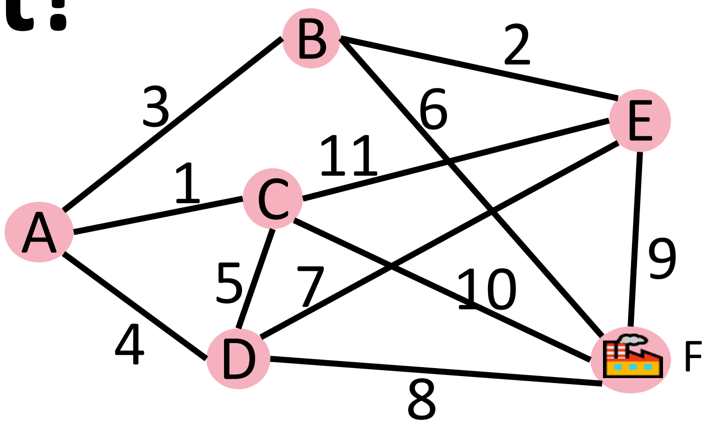


Node	known?	distTo	edgeTo
A		∞	
B		∞	
C		∞	
D		∞	
E		∞	
F		0	/

```
primMST(G graph, V start)
  Map edgeTo, distTo;
  initialize distTo to all ∞, 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)
```

Poll Everywhere pollev.com/uwcse373

Let's Try It!



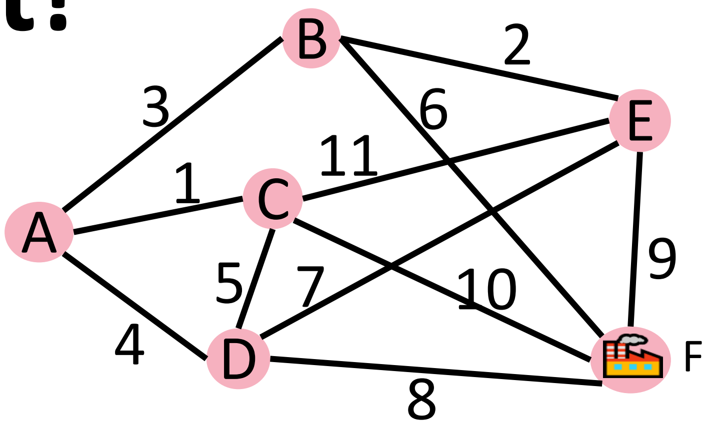
Pull F into the known set, updating its neighbors

Node	known?	distTo	edgeTo
A		∞	
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 ∞, 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)
```

Poll Everywhere pollev.com/uwcse373

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!

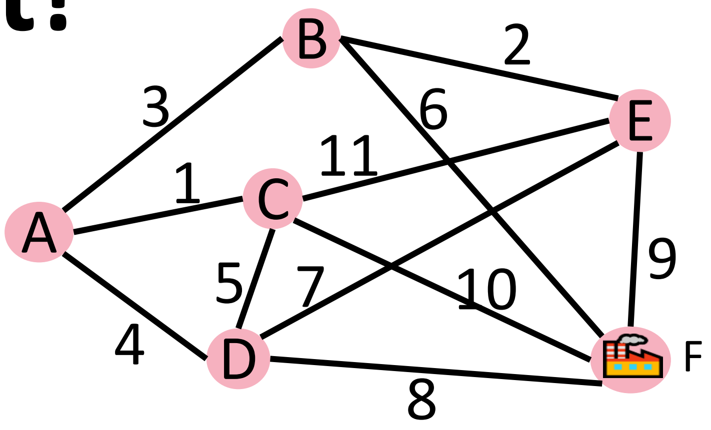
Node	known?	distTo	edgeTo
A		3??	B
B	Y	6	F
C		10??	F
D		8??	F
E		2??	B
F	Y	0	/

```
primMST(G graph, V start)
  Map edgeTo, distTo;
  initialize distTo to all ∞, 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)
```

Poll Everywhere

pollev.com/uwcse373

Let's Try It!



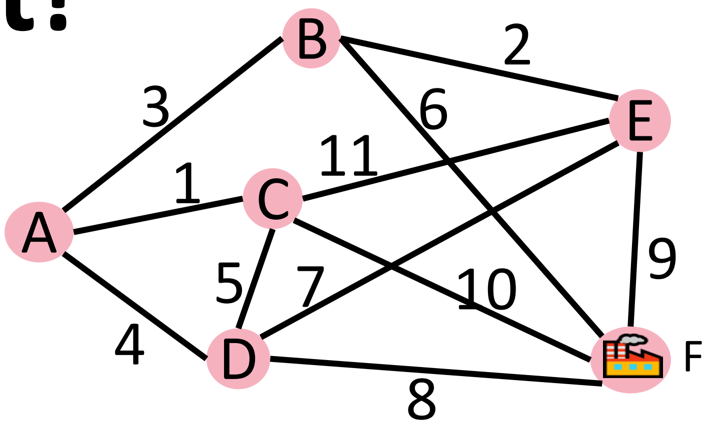
Choose E and update its neighbors. We found a smaller way to get to D!

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 ∞, 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)
```

Poll Everywhere pollev.com/uwcse373

Let's Try It!



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

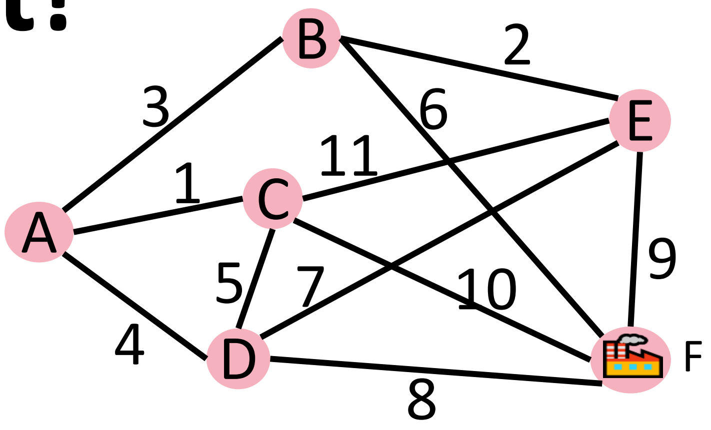
Node	known?	distTo	edgeTo
A	Y	3	B
B	Y	6	F
C		1??	A
D		4??	A
E	Y	2	B
F	Y	0	/

```
primMST(G graph, V start)
  Map edgeTo, distTo;
  initialize distTo to all ∞, 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)
```

Poll Everywhere pollev.com/uwcse373

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	/

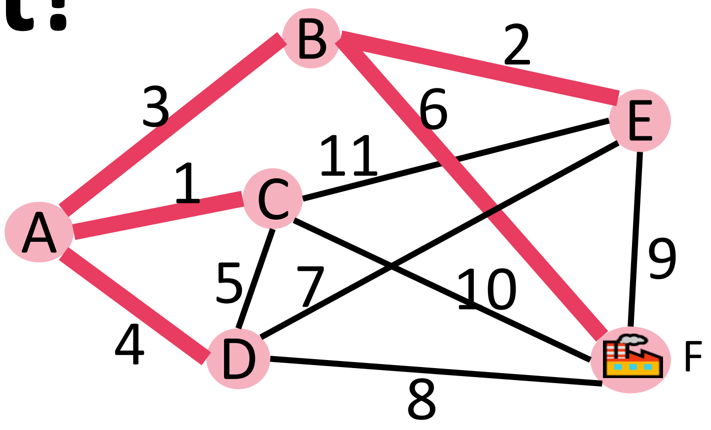
```
primMST(G graph, V start)
  Map edgeTo, distTo;
  initialize distTo to all ∞, except start to 0
  PriorityQueue<V> perimeter; add start;

  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)
```

Poll Everywhere

pollev.com/uwcse373

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 ∞, 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 Running

Final result:

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

Unsurprisingly, runtime is just like Dijkstra's algorithm.

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

$\Theta(|V|)$ iterations

$\Theta(\log|V|)$

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

total $\Theta(|E|)$ iterations

$\Theta(|V|)$

$\Theta(1)$

$\Theta(\log|V|)$

$\Theta(\log|V|)$

```
primMST(G graph, V start)
  Map edgeTo, distTo;
  initialize distTo with all nodes mappe

  PriorityQueue<V> perimeter; perimeter.

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

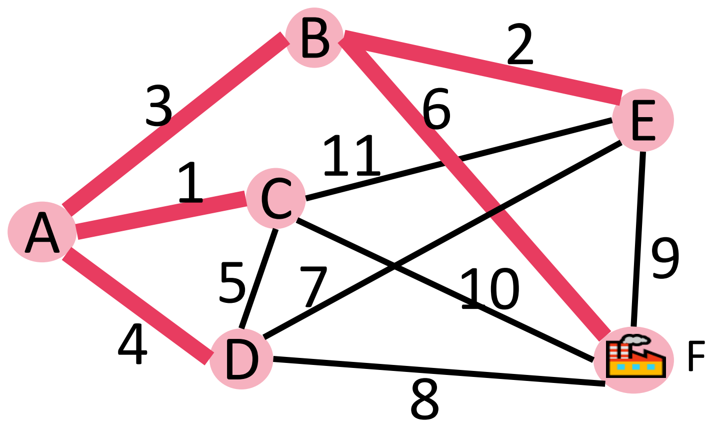
Poll Everywhere pollev.com/uwcse373

What if we started somewhere else?

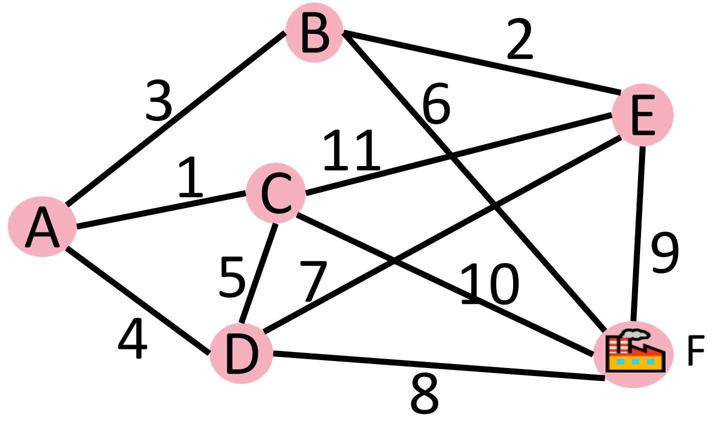
- In this example we started from the power plant, F. What would happen if we started from some other vertex in this graph?
  - a) We would no longer get a correct MST.
  - b) We would get an MST but it wouldn't solve the problem of connecting electricity.
  - c) We would get a correct MST, but a different one.
  - ✓ d) We would get the exact same MST.

- Since a Minimum Spanning Tree includes every vertex and minimizes all of its weights, it doesn't matter where we start!
  - This graph has a unique MST, but some graphs have multiple valid MSTs. In that case, starting elsewhere could give a different but correct MST!

Starting from F:



Starting from A? B? C?



Prim’s Demos and Visualizations

- Dijkstra’s Visualization
  - https://www.youtube.com/watch?v=1oiQ0hrVwJk
  - Dijkstra’s proceeds radially from its source, because it chooses edges by *path length from source*

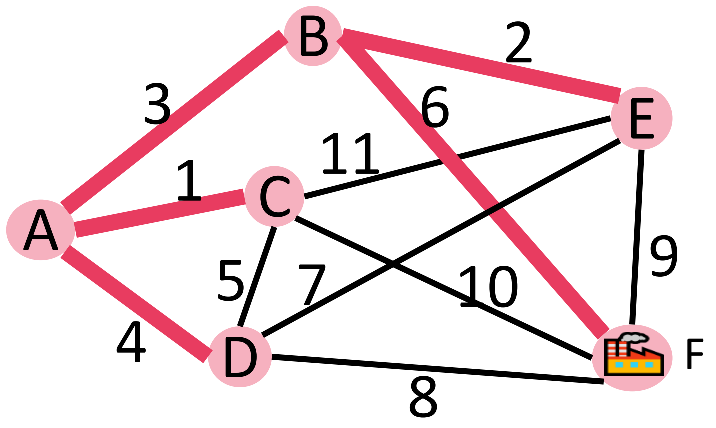
- Prim’s Visualization
  - https://www.youtube.com/watch?v=6uq0cQZOyoY
  - Prim’s jumps around the graph (the perimeter), because it chooses edges by *edge weight* (there’s no source)

Lecture Outline

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

A Different Approach

- **Observation**: We basically chose all the smallest edges in the entire graph (1, 2, 3, 4, 6)
  - The only exception was 5. Why didn't we add edge 5?
  - Because adding 5 would have created a cycle, and to connect A, C, & D we'd rather choose 1 & 4 than 1 & 5 or 4 & 5.



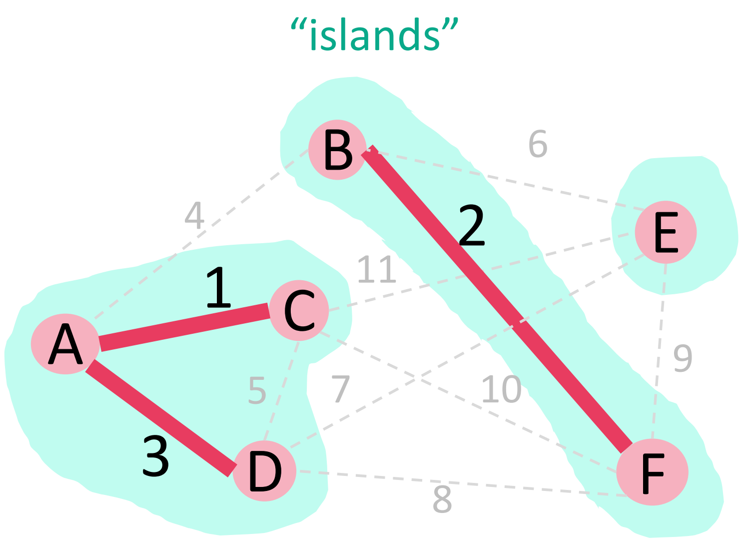
- Prim's thinks "vertex by vertex", but what if you think "edge by edge" instead?
  - Start with the smallest edge in the entire graph and work your way up
  - Add the edge to the MST as long as it connects two new groups (meaning don't add any edges that would create a cycle)

Building an MST "edge by edge" in this graph:

- Add edge 1
- Add edge 2
- Add edge 3
- Add edge 4
- Skip edge 5 (would create a cycle)
- Add edge 6
- Finished: all vertices in the MST!

Kruskal’s Algorithm

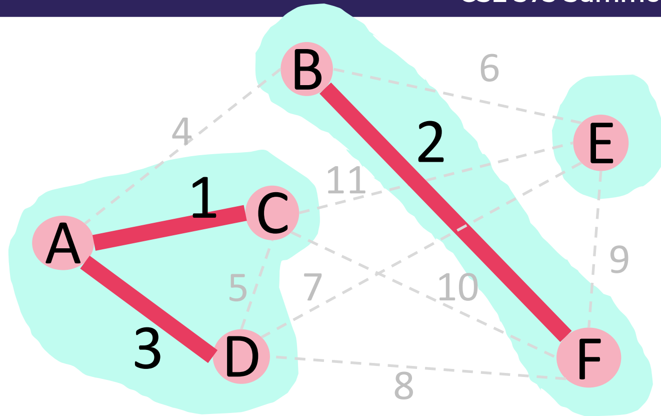
- This “edge by edge” approach is how **Kruskal’s Algorithm** works!
- Visualization: https://www.youtube.com/watch?v=ggLyKfBTABo



- **Key Intuition**: Kruskal’s keeps track of isolated “islands” of vertices (each is a sub-MST)
  - If an edge connects two vertices within the same “island”, it forms a cycle! Discard it.
  - If an edge connects two vertices in different “islands”, add it to the MST! Now those “islands” need to be combined.

```
kruskalMST(G graph)
  Set(?) msts; Set finalMST;
  initialize msts with each vertex as single-element MST
  sort all edges by weight (smallest to largest)

  for each edge (u,v) in ascending order:
    uMST = msts.find(u)
    vMST = msts.find(v)
    if (uMST != vMST):
      finalMST.add(edge (u, v))
      msts.union(uMST, vMST);
```

Disjoint Sets ADT (aka "Union-Find")



- Kruskal's needs to **find** what MST a vertex belongs to, and **union** those MSTs together

```
kruskalMST(G graph)
  DisjointSets<V> msts; Set finalMST;
  initialize msts with each vertex as single-element MST
  sort all edges by weight (smallest to largest)

  for each edge (u,v) in ascending order:
    uMST = msts.find(u)
    vMST = msts.find(v)
    if (uMST != vMST):
      finalMST.add(edge (u, v))
      msts.union(uMST, vMST);
```

DISJOINT SETS ADT

State

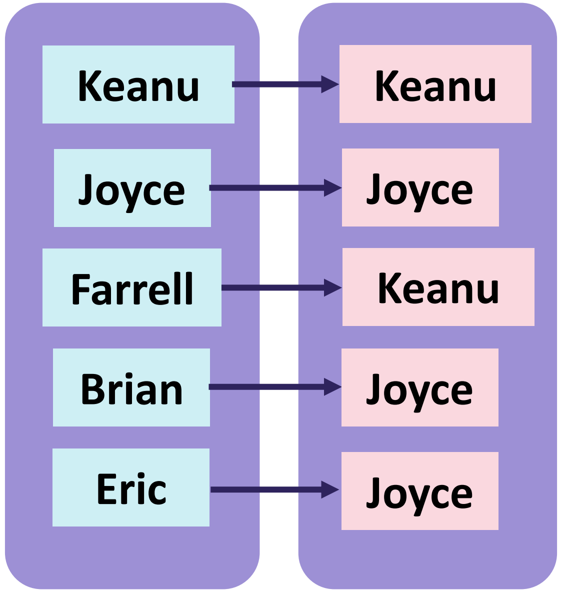
Family of Sets
- disjoint: no shared elements
- each set has a representative (use one of its members as a "name")

Behavior

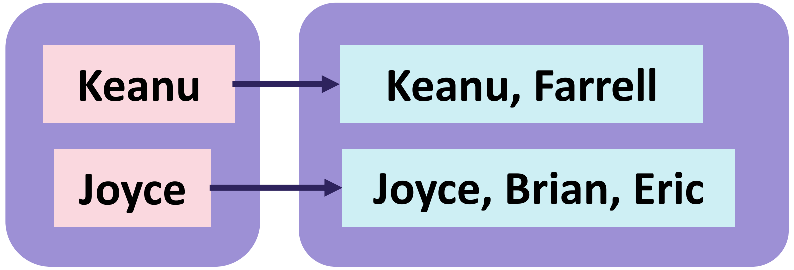
makeSet(value) – new set with value as only member (and representative)
findSet(value) – return representative of the set containing value
union(x, y) – combine sets containing x and y into one set with all elements, choose single new representative

Can we use an existing data structure?

Approach 1: map from value to representative element



Approach 2: map from representative to set of values



	Approach 1	Approach 2
makeSet(value)	$\Theta(1)$	$\Theta(1)$
findSet(value)	$\Theta(1)$	$\Theta(n)$
union(x, y)	$\Theta(n)$	$\Theta(n)$

DISJOINT SETS ADT

State

Family of Sets
- disjoint: no shared elements
- each set has a representative (use one of its members as a "name")

Behavior

makeSet(value) – new set with value as only member (and representative)

findSet(value) – return representative of the set containing value

union(x, y) – combine sets containing x and y into one set with all elements, choose single new representative

Both approaches limited by union: requires scanning through all elements to update. Could we do better? *Coming up next!*