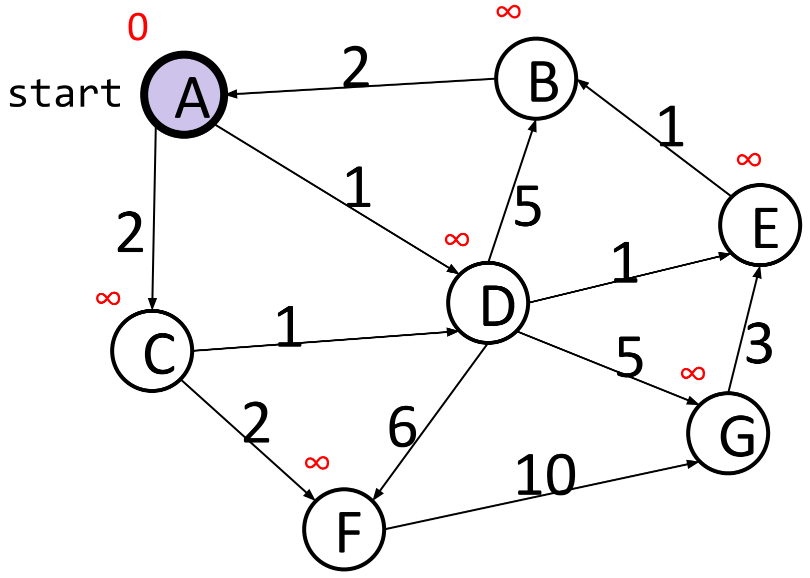
Lecture 18: MSTs

CSE 373: Data Structures and Algorithms

Dijkstra's Algorithm Warmup

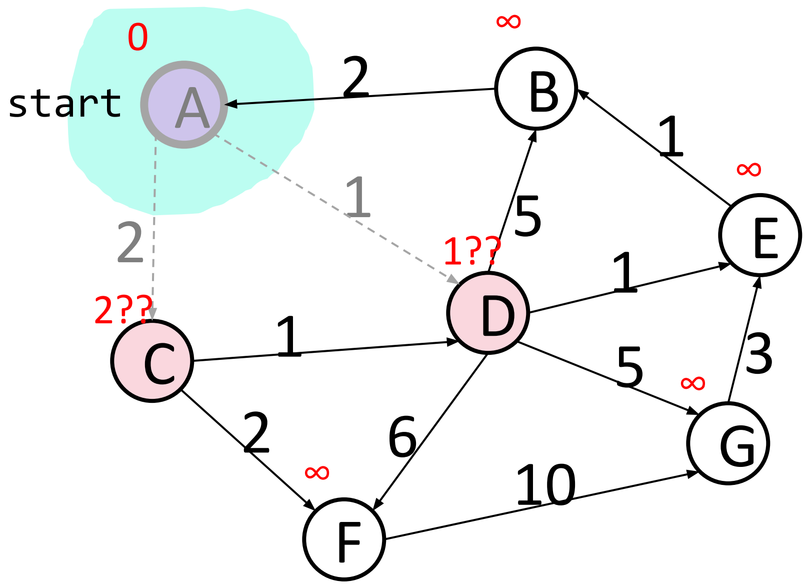
Slido Event #2633637
https://app.sli.do/event/kbACUWKZMpAbe41DJJZSWJ



Vertex	Known?	distTo	edgeTo
A		∞	
B		∞	
C		∞	
D		∞	
E		∞	
F		∞	
G		∞	

Order Added to Known Set:

Dijkstra's Algorithm Warmup

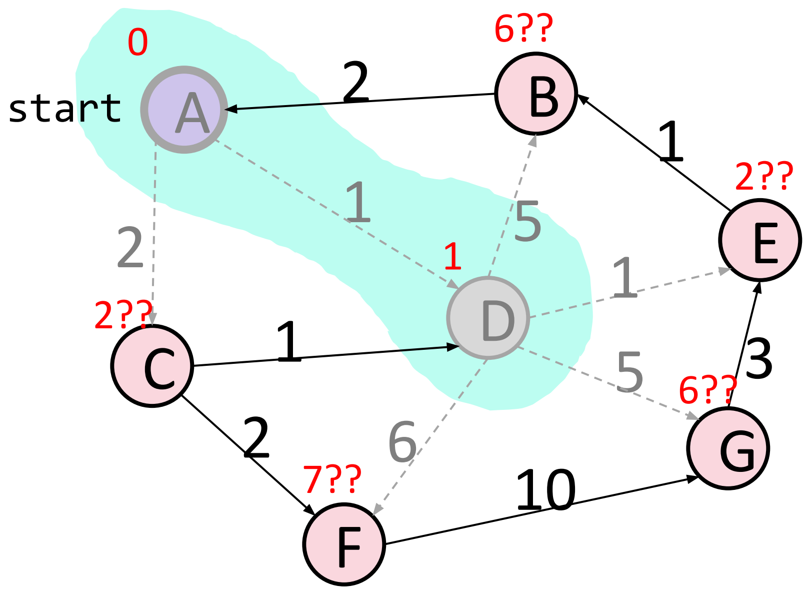


Order Added to Known Set:

A

Vertex	Known?	distTo	edgeTo
A	Y	0	/
B		∞	
C		≤ 2	A
D		≤ 1	A
E		∞	
F		∞	
G		∞	

Dijkstra's Algorithm Warmup

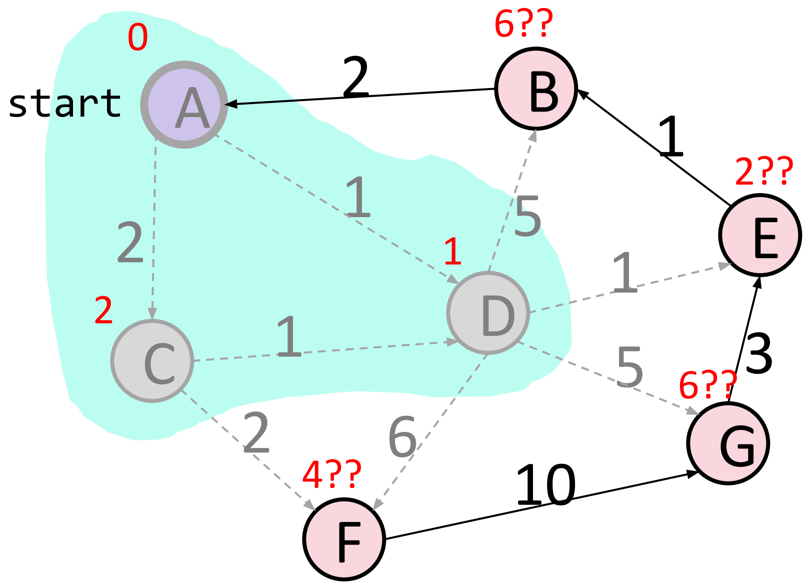


Vertex	Known?	distTo	edgeTo
A	Y	0	/
B		≤ 6	D
C		≤ 2	A
D	Y	1	A
E		≤ 2	D
F		≤ 7	D
G		≤ 6	D

Order Added to Known Set:

A, D

Dijkstra's Algorithm: Example #2

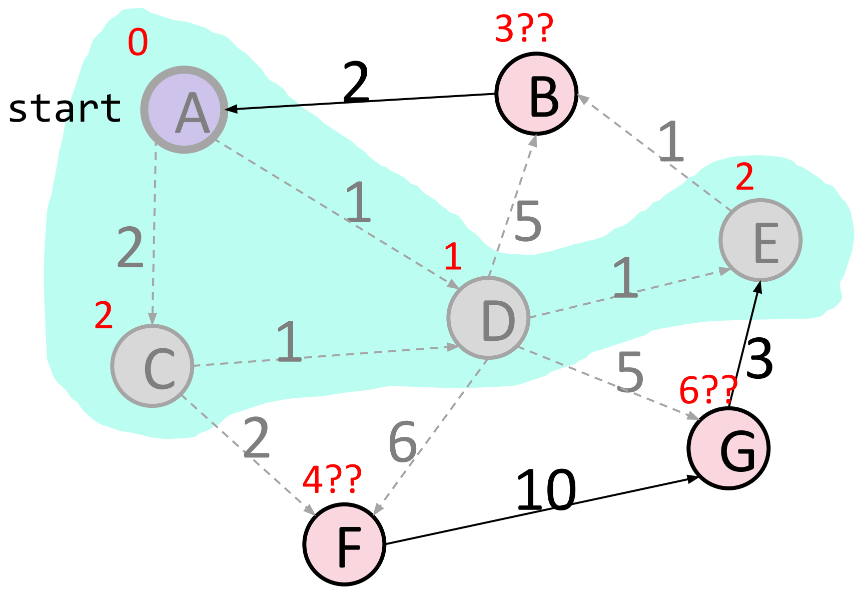


Vertex	Known?	distTo	edgeTo
A	Y	0	/
B		≤ 6	D
C	Y	2	A
D	Y	1	A
E		≤ 2	D
F		**≤ 4**	**C**
G		≤ 6	D

Order Added to Known Set:

A, D, C

Dijkstra's Algorithm: Example #2

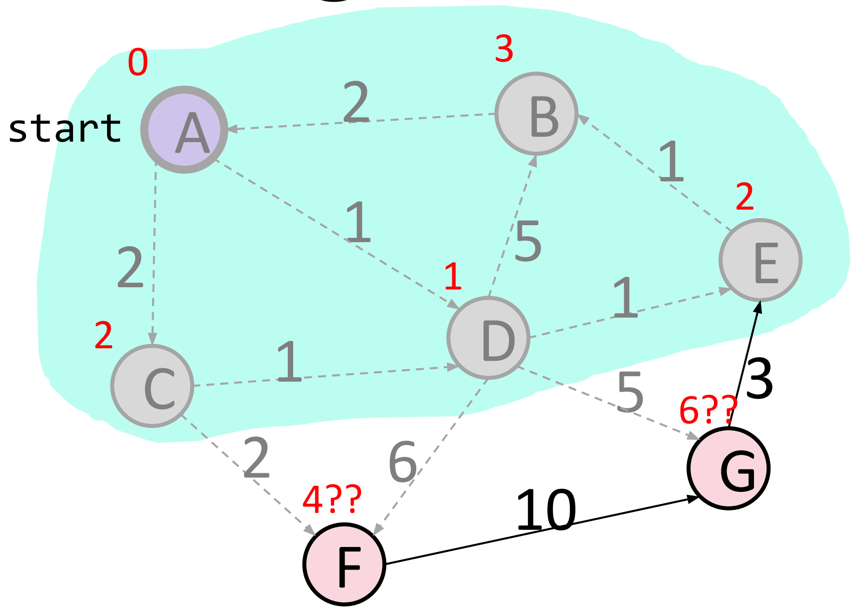


Vertex	Known?	distTo	edgeTo
A	Y	0	/
B		**≤ 3**	**E**
C	Y	2	A
D	Y	1	A
E	Y	2	D
F		≤ 4	C
G		≤ 6	D

Order Added to Known Set:

A, D, C, E

Dijkstra's Algorithm: Example #2

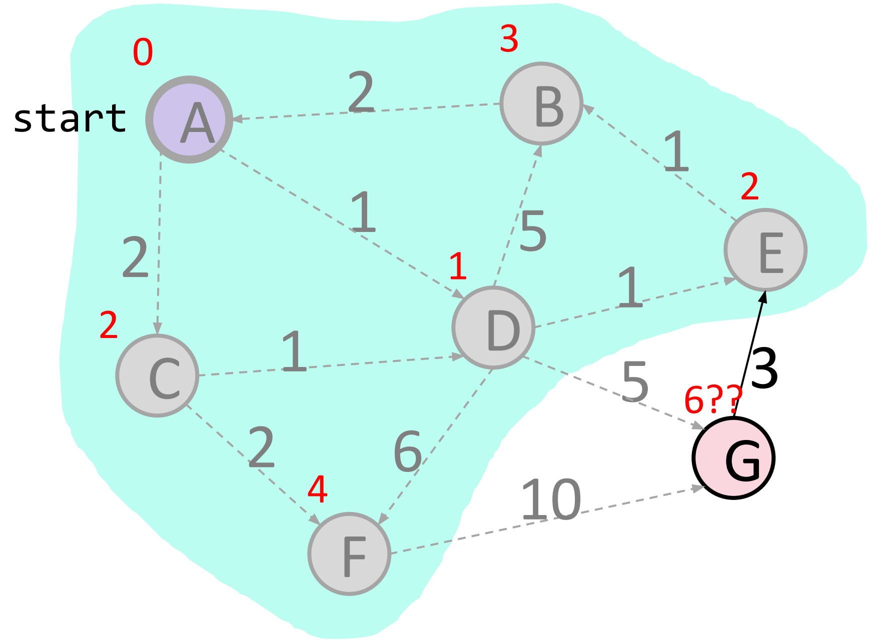


Order Added to Known Set:

A, D, C, E, B

Vertex	Known?	distTo	edgeTo
A	Y	0	/
B	Y	3	E
C	Y	2	A
D	Y	1	A
E	Y	2	D
F		≤ 4	C
G		≤ 6	D

Dijkstra's Algorithm: Example #2

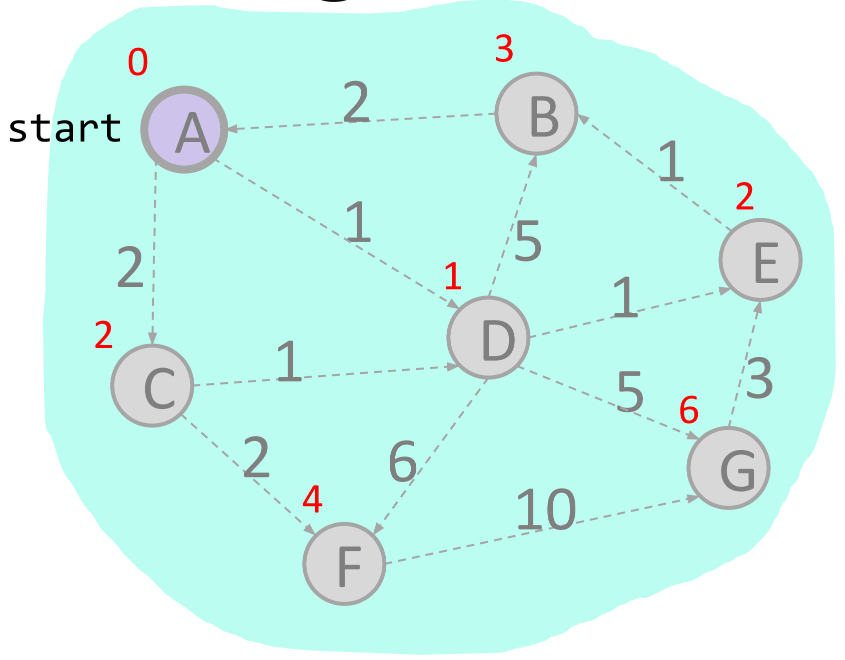


Order Added to Known Set:

A, D, C, E, B, F

Vertex	Known?	distTo	edgeTo
A	Y	0	/
B	Y	3	E
C	Y	2	A
D	Y	1	A
E	Y	2	D
F	Y	4	C
G		≤ 6	D

Dijkstra's Algorithm: Example #2



Order Added to Known Set:
A, D, C, E, B, F, G

Vertex	Known?	distTo	edgeTo
A	Y	0	/
B	Y	3	E
C	Y	2	A
D	Y	1	A
E	Y	2	D
F	Y	4	C
G	Y	6	D

Announcements

Midterm resubmission due Wednesday

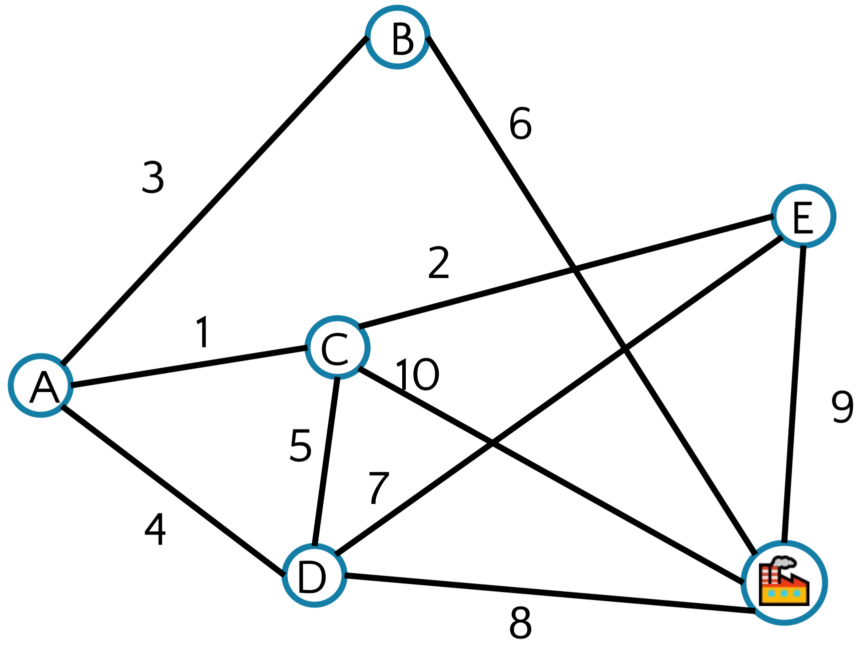
- NO LATE SUBMISSIONS

Minimum Spanning Trees

Prim's and Kruskal's Algorithms

Minimum Spanning Trees

It's the 1920's. Your friend at the electric company needs to choose where to build wires to connect all these cities to the plant.



She knows how much it would cost to lay electric wires between any pair of cities, and wants the cheapest way to make sure electricity runs from the plant to every city.

MST Problem

What do we need? A set of edges such that:

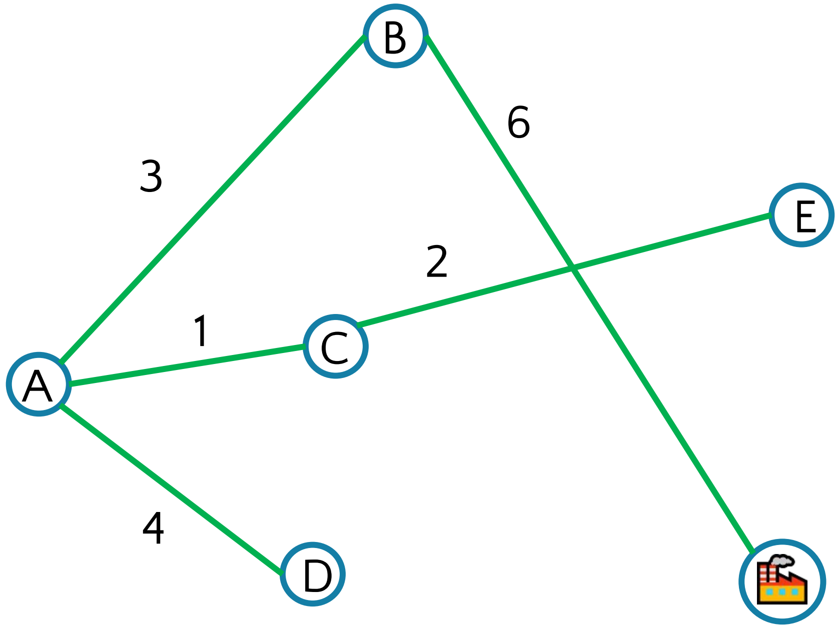
- Every vertex touches at least one of the edges. The edges "**span**" the graph.
- The graph on just those edges is **connected**.
- The minimum-weight set of edges that meet those conditions.

Claim: The set of edges we pick never has a cycle. Why?

MST is the exact number of edges to connect all vertices

- taking away 1 edge breaks connectedness
- adding 1 edge makes a cycle
- contains exactly V – 1 edges

Our result is a tree!



Minimum Spanning Tree Problem

Given: an undirected, weighted graph G
Find: A minimum-weight set of edges such that you can get from any vertex of G to any other on only those edges.

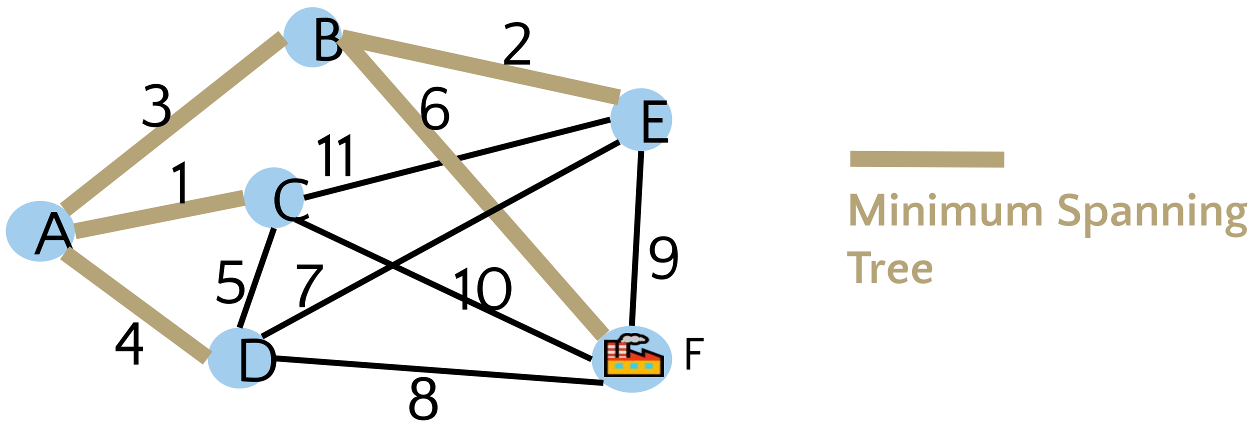
Question:

Is there always a unique MST for a given graph, yes or no?

Minimum Spanning Trees (MSTs)

A Minimum Spanning Tree for a graph is a set of that graph's edges that connect all of that graph's vertices (**spanning**) while minimizing the total weight of the set (**minimum**)

- Note: does NOT necessarily minimize the path from each vertex to every other vertex
- Any tree with V vertices will have V-1 edges
- A separate entity from the graph itself! More of an "annotation" applied to the graph, just like a Shortest Paths Tree (SPT)



Why do MST Algorithms Work?

- Two useful properties for MST edges. We can think about them from either perspective:
  - **Cycle Property**: The heaviest edge along a cycle is NEVER part of an MST.
  - **Cut Property**: Split the vertices of the graph into any two sets A and B. The lightest edge between A and B is ALWAYS part of an MST. *(Prim's thinks this way)*
- Whenever you add an edge to a tree you create exactly one cycle. Removing any edge from that cycle gives another tree!
- This observation, combined with the cycle and cut properties form the basis of all of the **greedy algorithms** for MSTs.
  - **greedy algorithm**: chooses best known option at each point and *commits*, rather than waiting for a global view of the graph before deciding

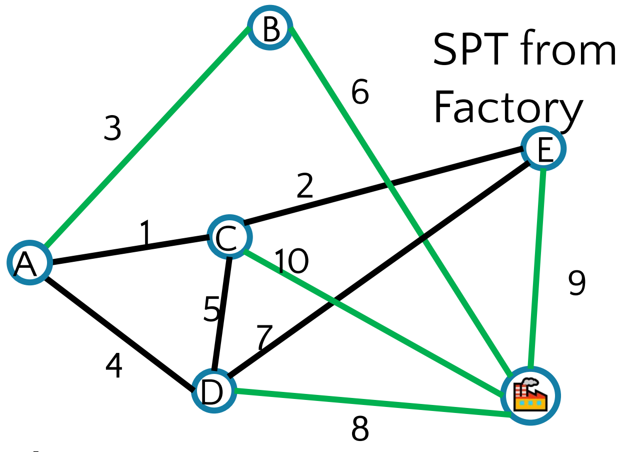
Shortest Path vs Minimum Spanning

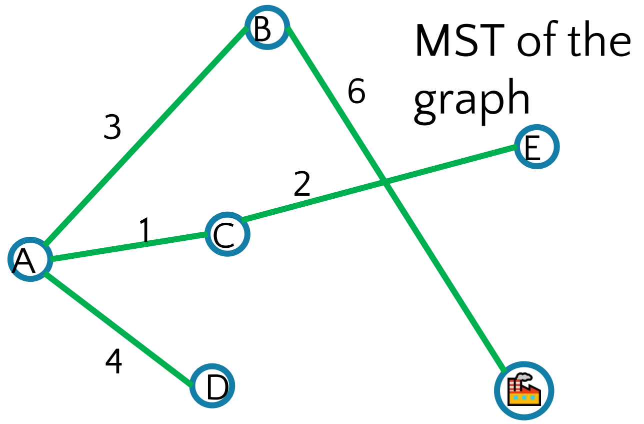
Shortest Path Problem

Given: a directed graph G and vertices s, t
Find: the shortest path from s to t.

Minimum Spanning Tree Problem

Given: an undirected, weighted graph G
Find: A minimum-weight set of edges such that you can get from any vertex of G to any other on only those edges.





Shortest Path Tree

- Specific start node (if you have a different start node, that changes the whole SPT, so there are multiple SPTs for graphs frequently)
- Keeps track of total path length

Minimum Spanning Tree

- No specific start node, since the goal is just to minimize the edge weights sum. Often only one possible MST that has the minimum sum
- All nodes connected
- Keeps track of cheapest edges that maintain connectivity

Minimum Spanning Trees

Prim’s and Kruskal’s Algorithms

Finding an MST

Here are two ideas for finding an MST:

Prim's

Think vertex-by-vertex

- Maintain a tree over a set of vertices
- Have each vertex remember the cheapest edge that could connect it to that set
- At every step, connect the vertex that can be connected the cheapest

Kruskal's

Think edge-by-edge

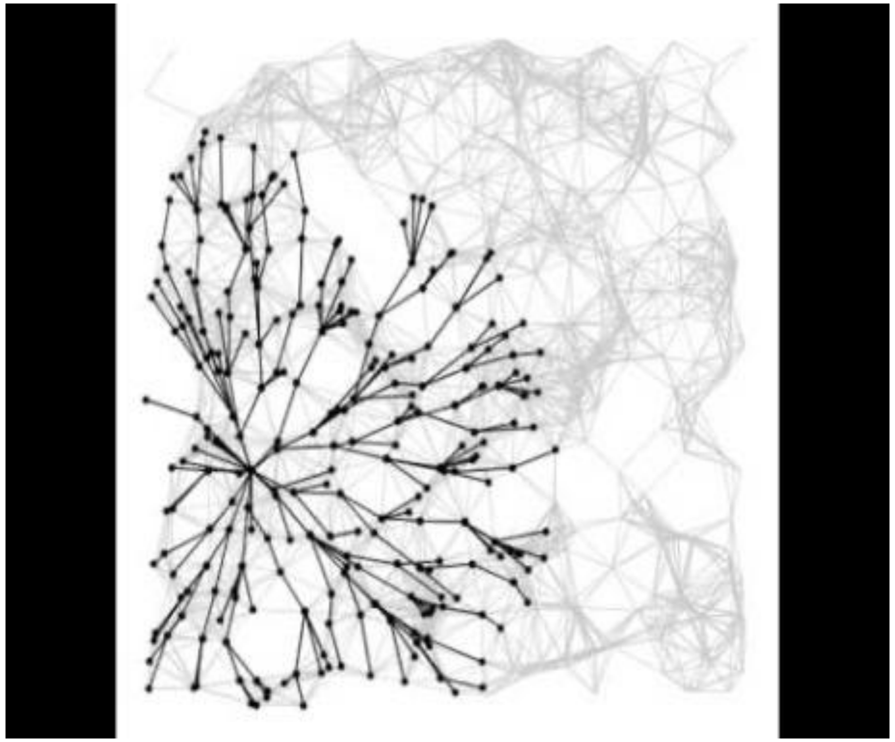
- Sort edges by weight. In increasing order:
    - add it if it connects new things to each other (don't add it if it would create a cycle)

Both ideas work!!

Dijkstra's versus Prim's

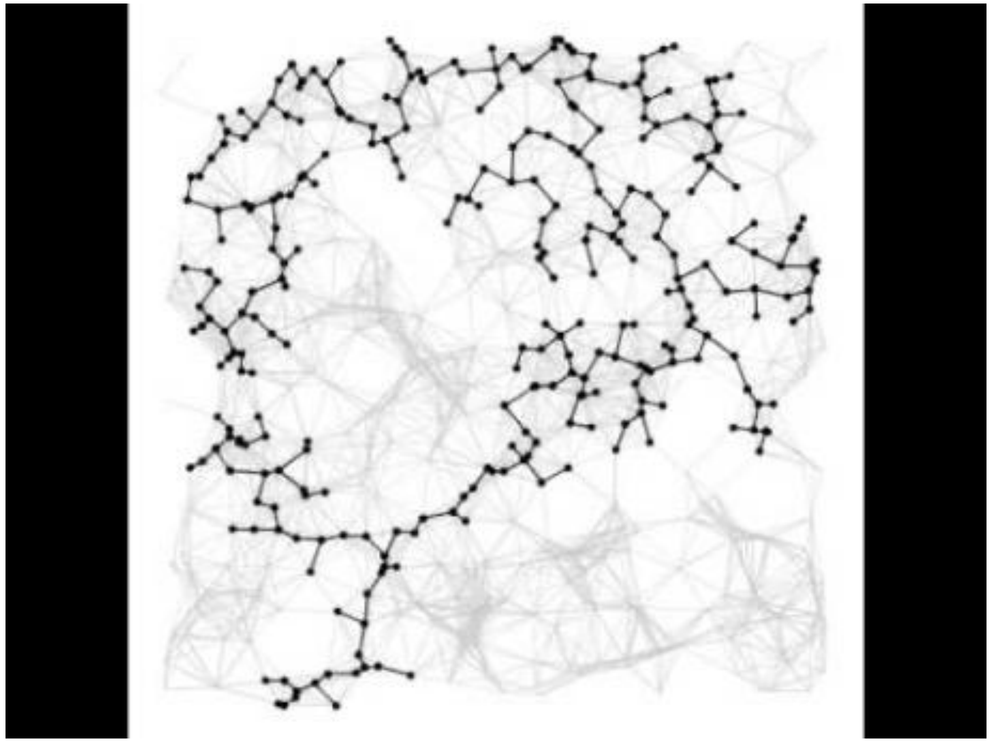
Dijkstra's Algorithm

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)



Prim's Algorithm

Question

Which lines of Dijkstra can we change to create our new algorithm?

Dijkstra's

1. Start at source
2. Update distance from current to unprocessed neighbors
3. Add closest unprocessed neighbor to solution
4. Repeat until all vertices have been marked processed

Algorithm idea:

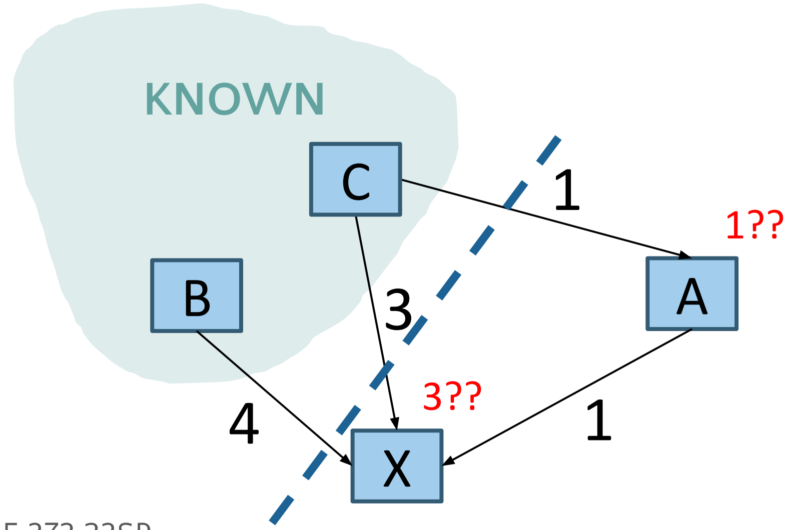
1. Start at any node
2. Investigate edges that connect unprocessed vertices
3. Add the lightest edge that grows connectivity to solution
4. Repeat until all vertices have been marked cessed

```
1Dijkstra(Graph G, Vertex source)
2   initialize distances to ∞
3   mark source as distance 0
4   mark all vertices unprocessed
5   while(there are unprocessed vertices){
5      let u be the closest unprocessed vertex
6      foreach(edge (u,v) leaving u){
7         if(u.dist+weight(u,v) < v.dist){
8            v.dist = u.dist+weight(u,v)
9            v.predecessor = u
10        }
11     }
12     mark u as processed
13  }
14
15
```

```
Prims(Graph G, Vertex source)
2   initialize distances to ∞
3   mark source as distance 0
4   mark all vertices unprocessed
5   while(there are unprocessed vertices){
5      let u be the closest unprocessed vertex
6      foreach(edge (u,v) leaving u){
7         if(weight(u,v) < v.dist){
8            v.dist = u.dist+weight(u,v)
9            v.predecessor = u
10        }
11     }
12     mark u as processed
13  }
14
15
```

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?"



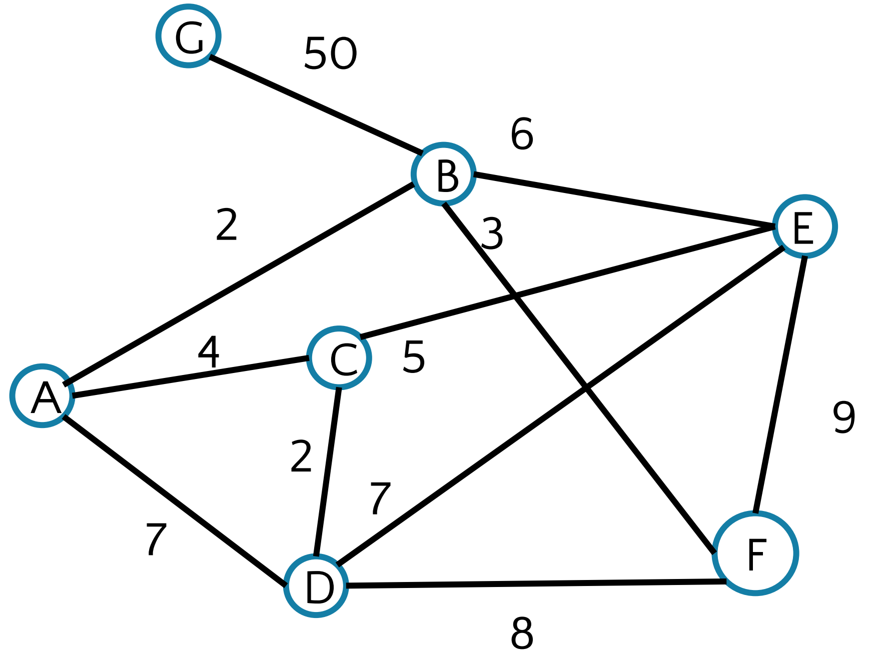
```
primsShortestPath(G graph, V start)
  Map edgeTo, distTo;
  initialize distTo with all nodes mapped to ∞, except start to 0
  PriorityQueue<V> perimeter; perimeter.add(start);

  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(u, newDist)
        edgeTo.put(u, v)
        if (perimeter.contains(v)):
          perimeter.changePriority(v, newDist)
        else:
          perimeter.add(v, newDist)
```

Try It Out

```
primsShortestPath(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(u, newDist)
        edgeTo.put(u, v)
        if (perimeter.contains(v)):
          perimeter.changePriority(v, newDist)
        else:
          perimeter.add(v, newDist)
```



Vertex	Distance	Best Edge	Processed
A	-	X	✓
B	2	(A, B)	✓
C	4	(A, C)	✓
D	2	(C, D)	✓
E	5	(C, E)	✓
F	3	(B, F)	✓
G	50	(B, G)	✓

A Different Approach

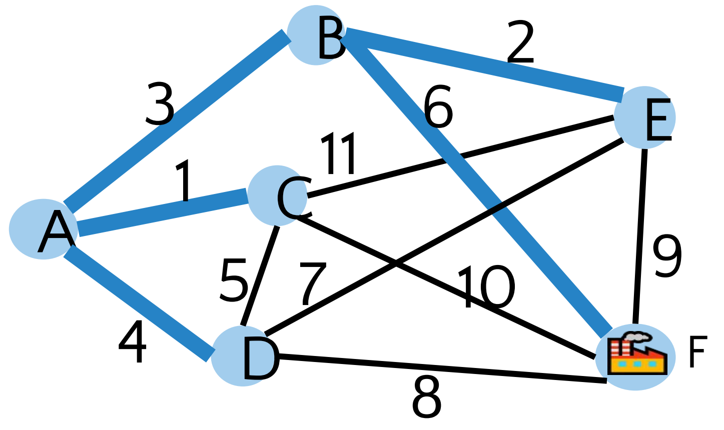
Suppose the MST on the right was produced by Prim's

Observation: We basically choose all the smallest edges in the entire graph (1, 2, 3, 4, 6)

- The only exception was 5. Why shouldn't we add edge 5?
- Because adding 5 would create 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)

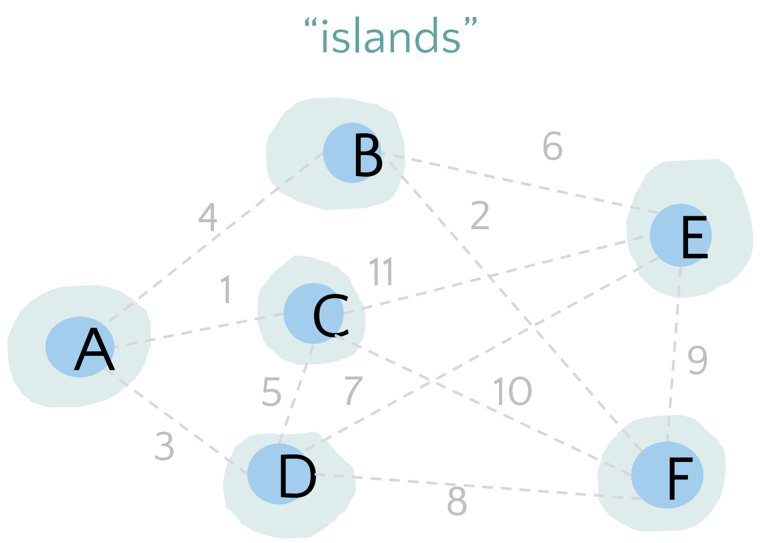


Building an MST "edge by edge" in this graph:

- 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!



Key Intuition: Kruskal's keeps track of isolated "islands" of vertices (each is a sub-MST)

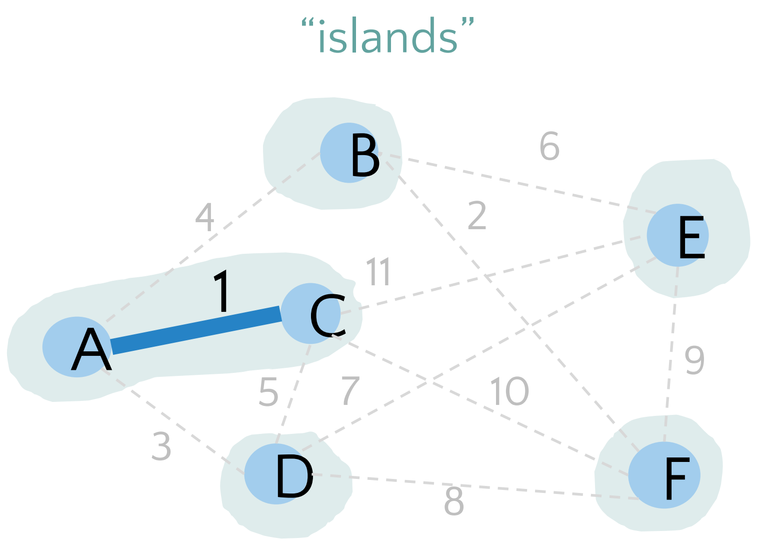
- Start with each vertex as its own "island"
- If an edge connects two vertices within the same "island", it forms a cycle! Discard it.
- If an edge connects two vertices in different "islands", add it to the MST! Now those "islands" need to be combined.

```
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)
```

Kruskal's Algorithm

This "edge by edge" approach is how **Kruskal's Algorithm** works!



Key Intuition: Kruskal's keeps track of isolated "islands" of vertices (each is a sub-MST)

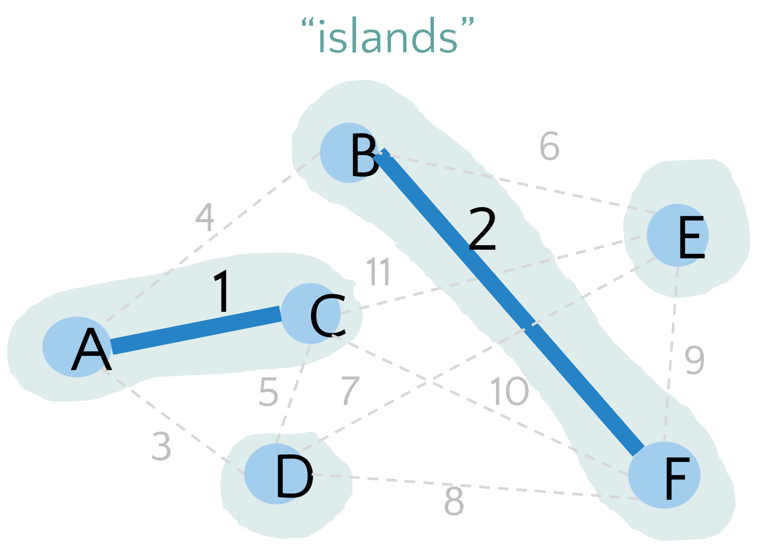
- Start with each vertex as its own "island"
- If an edge connects two vertices within the same "island", it forms a cycle! Discard it.
- If an edge connects two vertices in different "islands", add it to the MST! Now those "islands" need to be combined.

```
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)
```

Kruskal's Algorithm

This "edge by edge" approach is how **Kruskal's Algorithm** works!



Key Intuition: Kruskal's keeps track of isolated "islands" of vertices (each is a sub-MST)

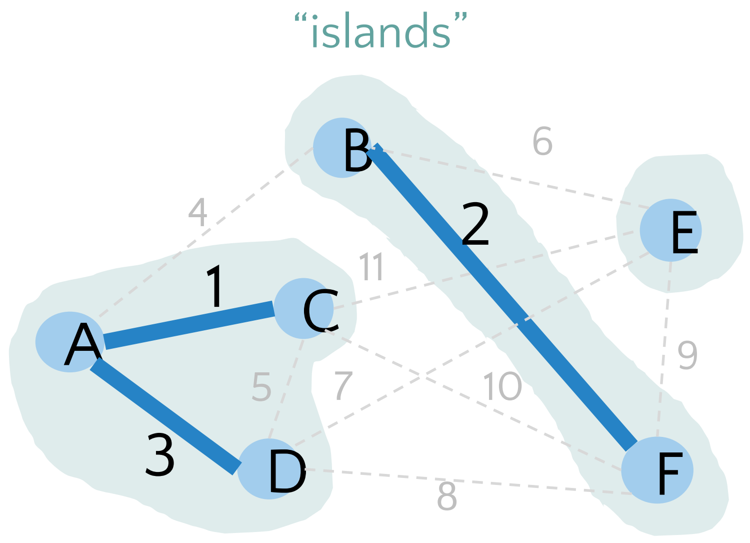
- Start with each vertex as its own "island"
- If an edge connects two vertices within the same "island", it forms a cycle! Discard it.
- If an edge connects two vertices in different "islands", add it to the MST! Now those "islands" need to be combined.

```
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)
```

Kruskal's Algorithm

This "edge by edge" approach is how **Kruskal's Algorithm** works!



Key Intuition: Kruskal's keeps track of isolated "islands" of vertices (each is a sub-MST)

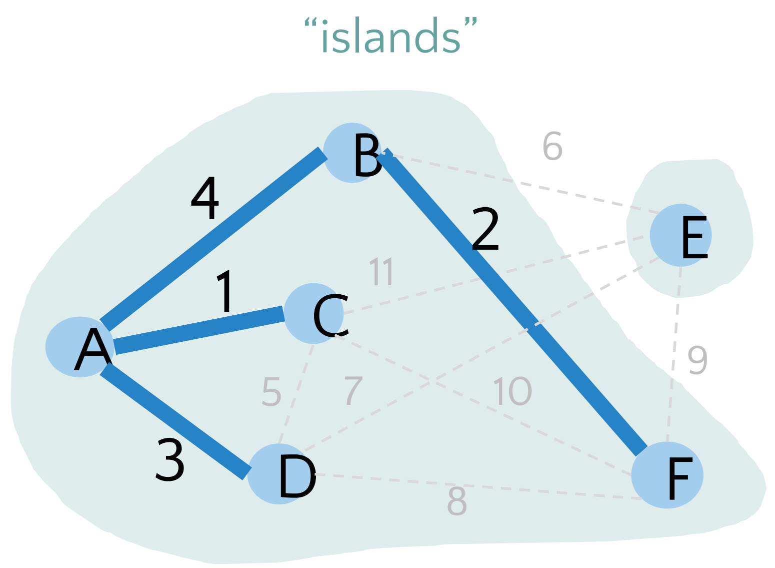
- Start with each vertex as its own "island"
- If an edge connects two vertices within the same "island", it forms a cycle! Discard it.
- If an edge connects two vertices in different "islands", add it to the MST! Now those "islands" need to be combined.

```
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)
```

Kruskal's Algorithm

This "edge by edge" approach is how **Kruskal's Algorithm** works!



Key Intuition: Kruskal's keeps track of isolated "islands" of vertices (each is a sub-MST)

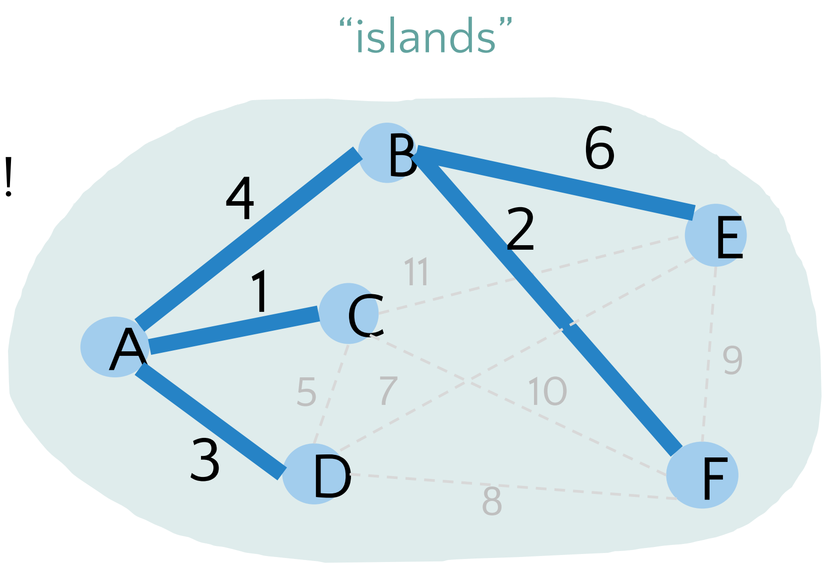
- Start with each vertex as its own "island"
- If an edge connects two vertices within the same "island", it forms a cycle! Discard it.
- If an edge connects two vertices in different "islands", add it to the MST! Now those "islands" need to be combined.

```
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)
```

Kruskal's Algorithm

This "edge by edge" approach is how **Kruskal's Algorithm** works!



Key Intuition: Kruskal's keeps track of isolated "islands" of vertices (each is a sub-MST)

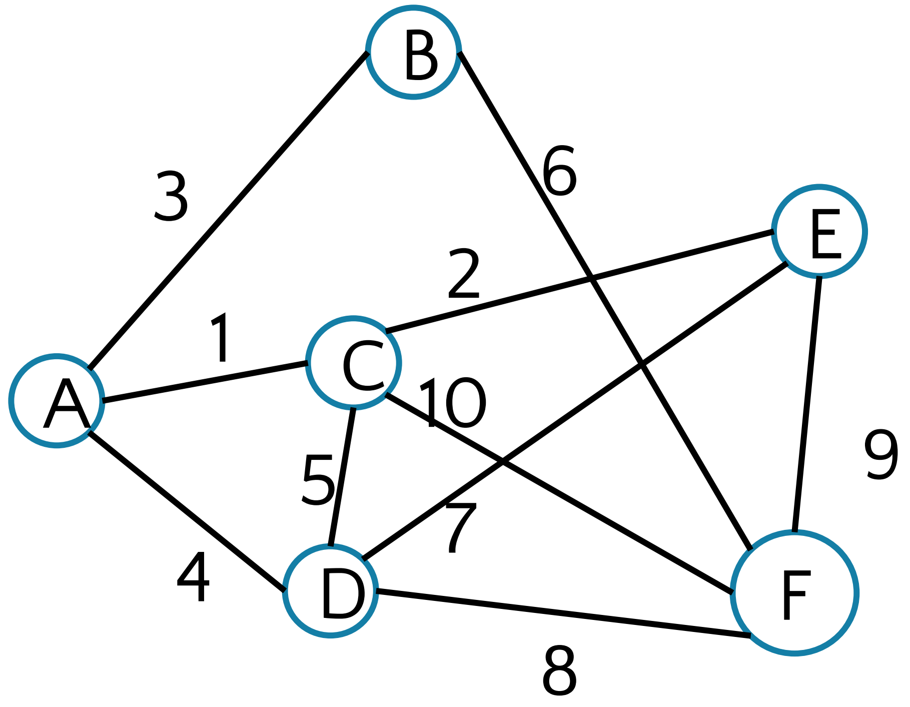
- Start with each vertex as its own "island"
- If an edge connects two vertices within the same "island", it forms a cycle! Discard it.
- If an edge connects two vertices in different "islands", add it to the MST! Now those "islands" need to be combined.

```
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)
```

Try It Out

```
KruskalMST(Graph G)
    initialize each vertex to be its own component
    sort the edges by weight
    foreach(edge (u, v) in sorted order){
        if(u and v are in different components){
            add (u,v) to the MST
            Update u and v to be in the same component
        }
    }
```

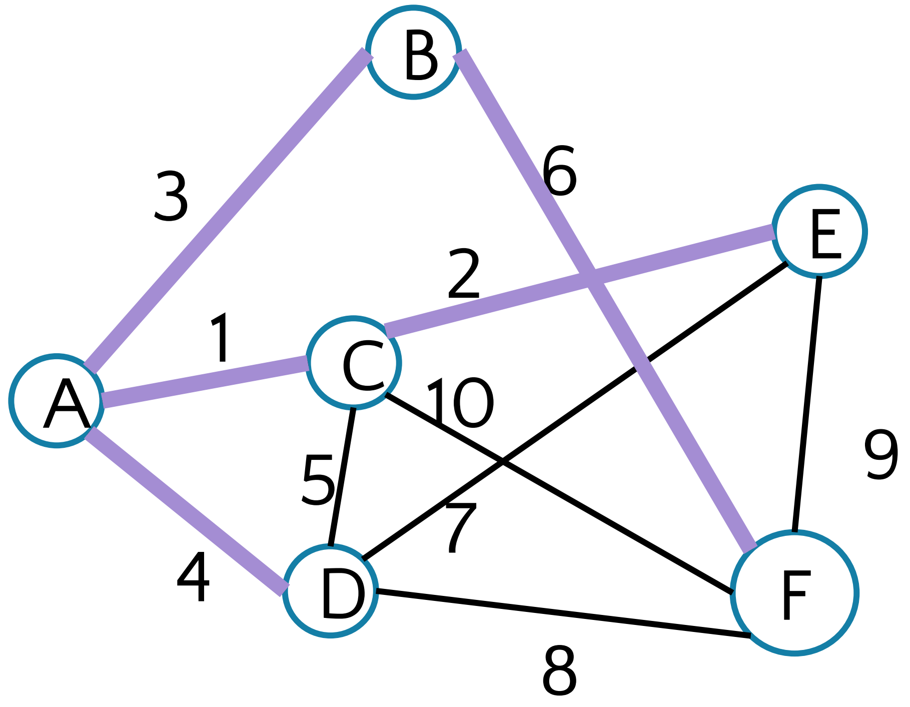


Edge	Include?	Reason
(A,C)		
(C,E)		
(A,B)		
(A,D)		
(C,D)		

Edge (cont.)	Inc?	Reason
(B,F)		
(D,E)		
(D,F)		
(E,F)		
(C,F)		

Try It Out

```
KruskalMST(Graph G)
    initialize each vertex to be its own component
     sort the edges by weight
     foreach(edge (u, v) in sorted order){
         if(u and v are in different components){
              add (u,v) to the MST
              Update u and v to be in the same component
         }
     }
```



Edge	Include?	Reason
(A,C)	Yes	
(C,E)	Yes	
(A,B)	Yes	
(A,D)	Yes	
(C,D)	No	Cycle A,C,D,A

Edge (cont.)	Inc?	Reason
(B,F)	Yes	
(D,E)	No	Cycle A,C,E,D,A
(D,F)	No	Cycle A,D,F,B,A
(E,F)	No	Cycle A,C,E,F,D,A
(C,F)	No	Cycle C,A,B,F,C

Kruskal’s Implementation

```
KruskalMST(Graph G)
   initialize each vertex to be its own component
    sort the edges by weight
    foreach(edge (u, v) in sorted order){
        if(u and v are in different components){
            add (u,v) to the MST
            Update u and v to be in the same component
        }
    }
```

Some lines of code there were a little sketchy.

- `initialize each vertex to be its own component`
- `Update u and v to be in the same component`

Can we use one of our data structures?

Questions?

That's all!