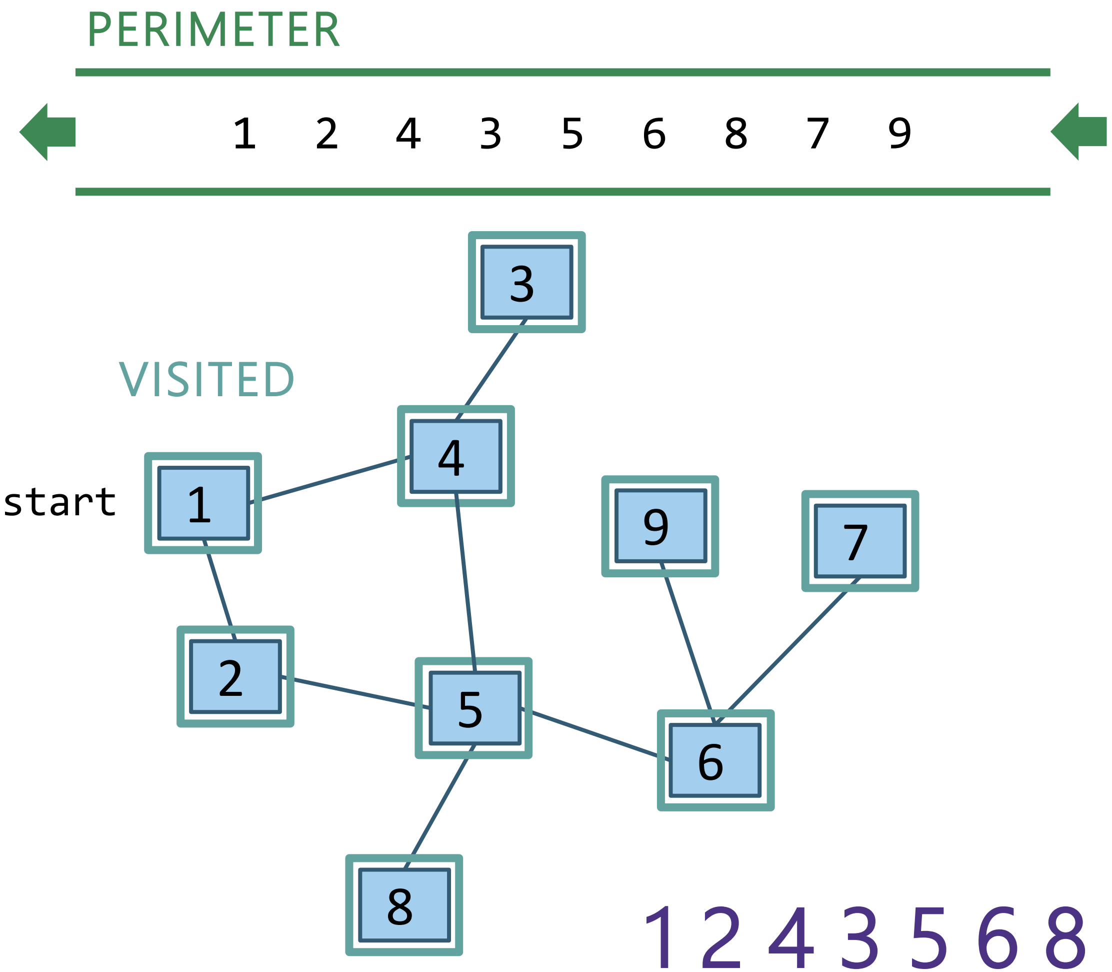
Lecture 18: MSTs

CSE 373: Data Structures and Algorithms

Warm Up - BFS

Give a possible ordering of a BFS traversal of the following graph. Break ties between unvisited vertices by visiting the smaller vertex first



1 2 4 3 5 6 8 7 9

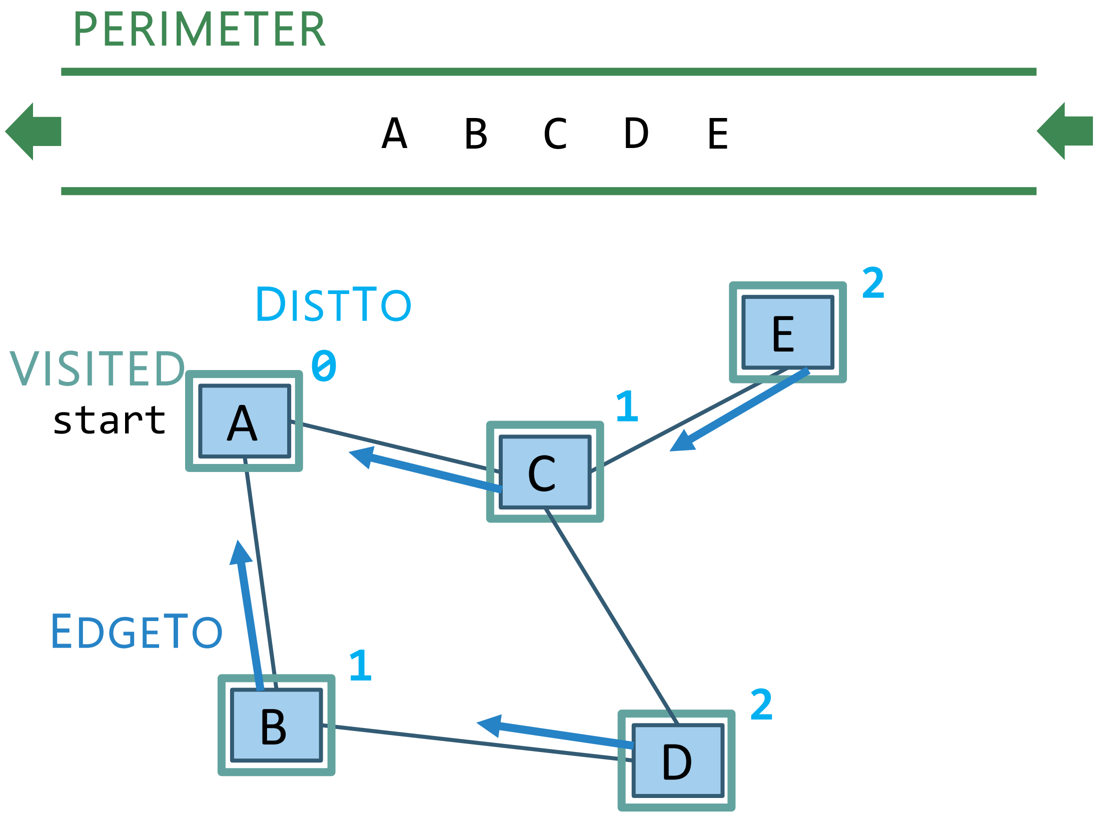
```
bfs(Graph graph, Vertex start) {
  Queue<Vertex> perimeter = new Queue<>();
  Set<Vertex> visited = new Set<>();

  perimeter.add(start);
  visited.add(start);

  while (!perimeter.isEmpty()) {
    Vertex from = perimeter.remove();
    for (Edge edge : graph.edgesFrom(from)) {
      Vertex to = edge.to();
      if (!visited.contains(to)) {
        perimeter.add(to);
        visited.add(to);
      }
    }
  }
}
```

Administrivia

- Midterm due TONIGHT at 11:59pm – NO LATE ASSIGNMENTS
- Q4.1
  - you can assume that going either left or right cuts the value of N in half
- Q7.1
  - Files have a unique memory address
  - You may select to chop up the String if you choose in your design
- Kasey midterm "OH" tonight
  - will be hanging out in Discord OH Lobby and monitoring Ed board tonight from 7pm on to clarify any last-minute questions

- Exercise 3 comes out later today

BFS for Shortest Paths: Example



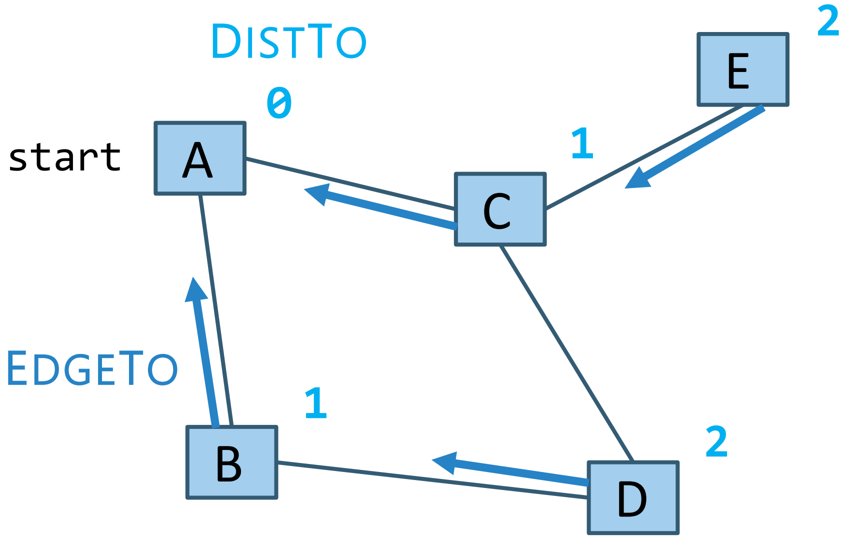
The **edgeTo** map stores backpointers: each vertex remembers what vertex was used to arrive at it!

Note: this code stores **visited**, **edgeTo**, and **distTo** as **external maps** (only drawn on graph for convenience). Another implementation option: store them as fields of the nodes themselves

```
  ...
  Map<Vertex, Edge> edgeTo = ...
  Map<Vertex, Double> distTo = ...

  edgeTo.put(start, null);
  distTo.put(start, 0.0);

  while (!perimeter.isEmpty()) {
    Vertex from = perimeter.remove();
    for (Edge edge : graph.edgesFrom(from)) {
      Vertex to = edge.to();
      if (!visited.contains(to)) {
        edgeTo.put(to, edge);
        distTo.put(to, distTo.get(from) + 1);
        perimeter.add(to);
        visited.add(to);
      }
    }
  }
  return edgeTo;
}
```

What about the Target Vertex?

Shortest Path Tree:



This modification on BFS didn't mention the target vertex at all!

Instead, it calculated the shortest path and distance from start to *every other vertex*

- This is called the shortest path tree
  - A general concept: in this implementation, made up of distances and backpointers

Shortest path tree has all the answers!

- Length of shortest path from A to D?
  - Lookup in distTo map: 2
- What's the shortest path from A to D?
  - Build up backwards from edgeTo map: start at D, follow backpointer to B, follow backpointer to A – our shortest path is A → B → D

All our shortest path algorithms will have this property

- If you only care about t, you can sometimes stop early!

Recap: Graph Problems

Just like everything is Graphs, every problem is a Graph Problem

BFS and DFS are very useful tools to solve these! We'll see plenty more.

EASY

MEDIUM

s-t Connectivity Problem
Given source vertex **s** and a target vertex **t**, does there exist a path between **s** and **t**?

BFS or DFS + check if we've hit t

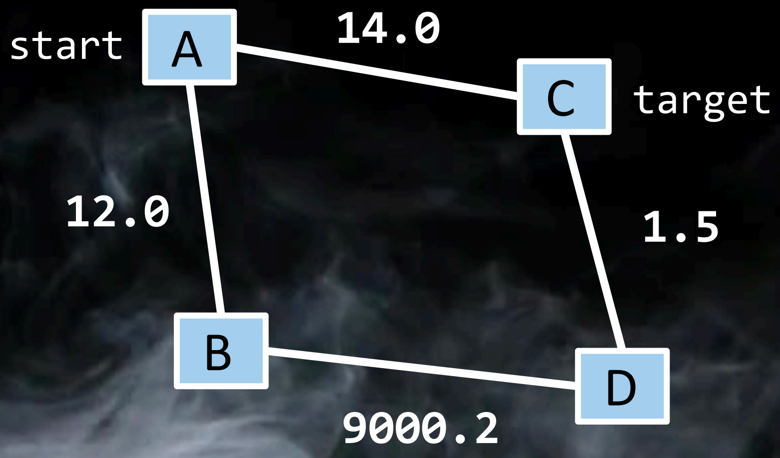
(Unweighted) Shortest Path Problem
Given source vertex **s** and a target vertex **t**, how long is the shortest path from **s** to **t**? What edges make up that path?

BFS + generate shortest path tree as we go

What about the Shortest Path Problem on a *weighted* graph?

Next Stop Weighted Shortest Paths

HARDER (FOR NOW)



- Suppose we want to find shortest path from A to C, using weight of each edge as "distance"
- Is BFS going to give us the right result here?

Dijkstra’s Algorithm

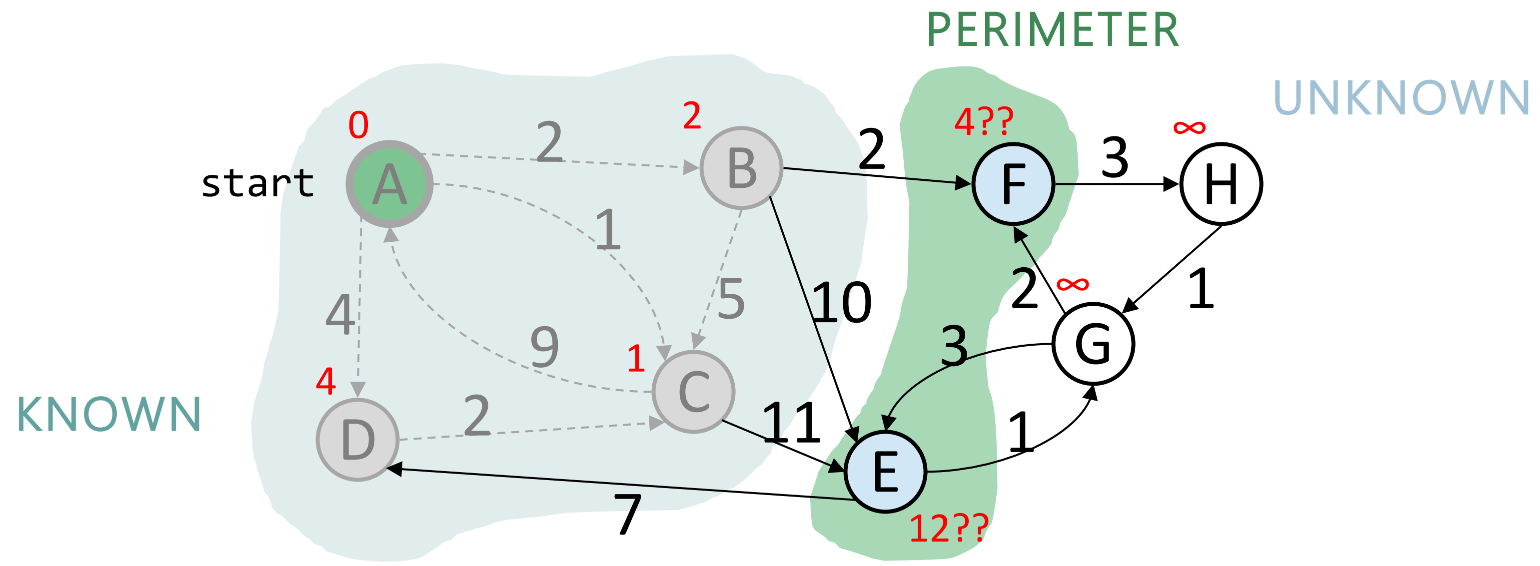
Named after its inventor, Edsger Dijkstra (1930-2002)

- Truly one of the “founders” of computer science
- 1972 Turing Award
- This algorithm is just *one* of his many contributions!
- Example quote: “Computer science is no more about computers than astronomy is about telescopes”

The idea: reminiscent of BFS, but adapted to handle weights

- Grow the set of nodes whose shortest distance has been computed
- Nodes not in the set will have a “best distance so far”

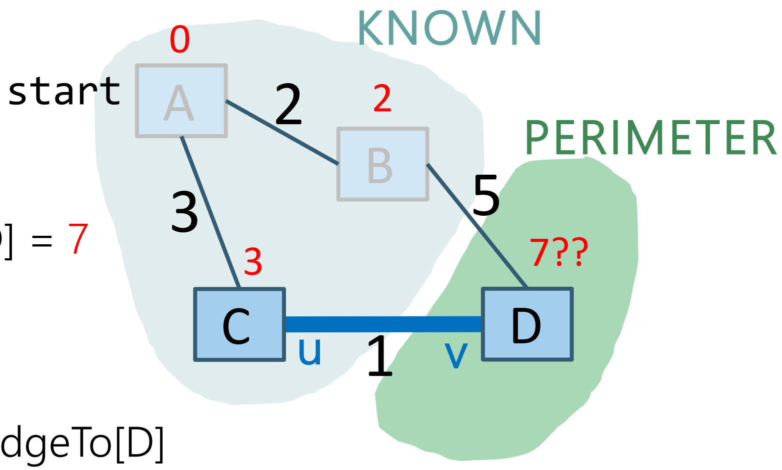
Dijkstra's Algorithm: Idea



Initialization:
- Start vertex has distance 0; all other vertices have distance ∞

At each step:
- Pick closest unknown vertex v
- Add it to the "cloud" of known vertices
- Update "best-so-far" distances for vertices with edges from v

Dijkstra's Pseudocode (High-Level)



- Suppose we already visited B, distTo[D] = 7
- Now considering edge (C, D):
  - `oldDist = 7`
  - `newDist = 3 + 1`
  - That's better! Update distTo[D], edgeTo[D]

Similar to "visited" in BFS, "known" is nodes that are finalized (we know their path)

Dijkstra's algorithm is all about updating "best-so-far" in distTo if we find shorter path! Init all paths to infinite.

Order matters: always visit closest first!

Consider all vertices reachable from me: would getting there *through* me be a shorter path than they currently know about?

```
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) from u 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)
```

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

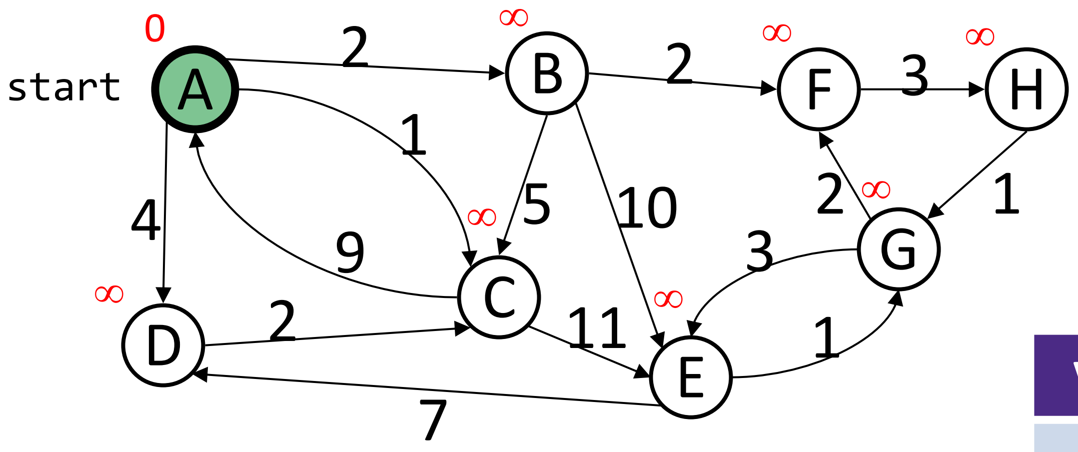
- We call this update relaxing the distance because it only ever shortens the current best path

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)
```

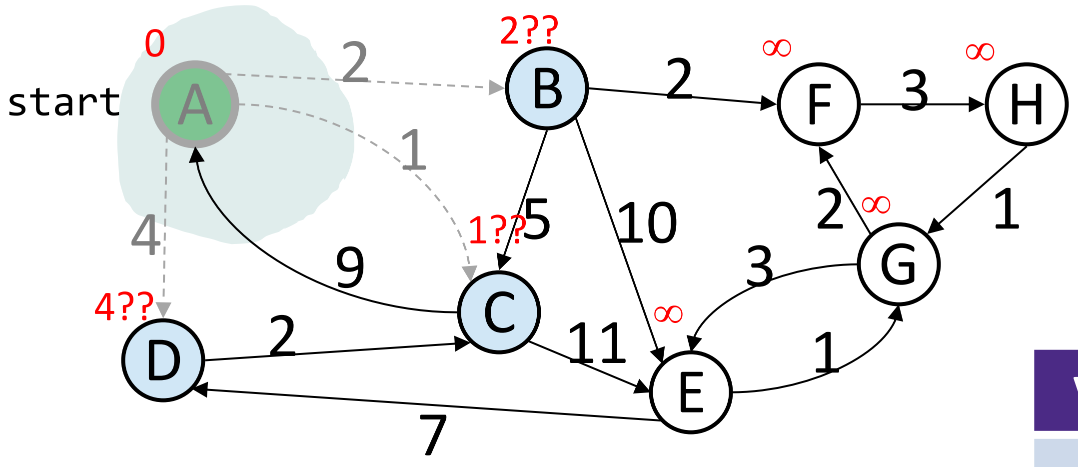
Dijkstra's Algorithm: Example #1



Order Added to Known Set:

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

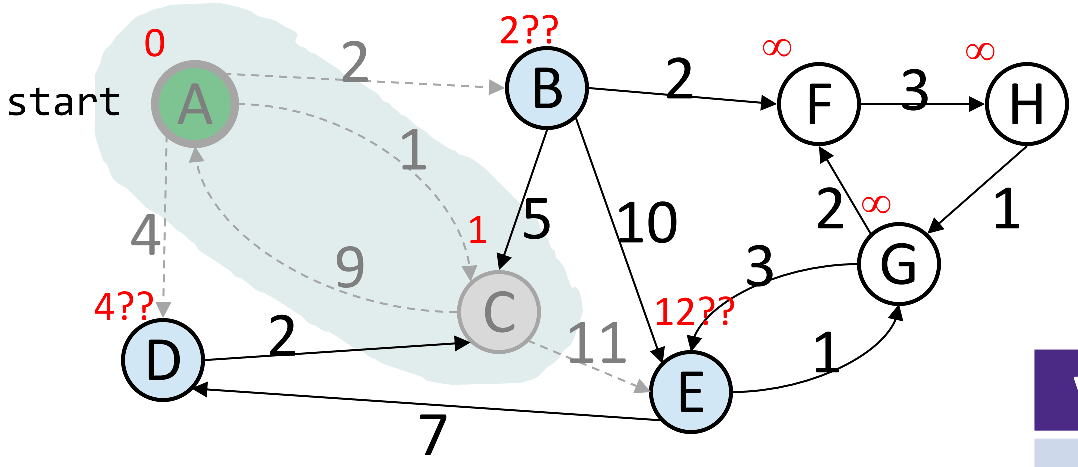
Dijkstra's Algorithm: Example #1



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

<u>Order Added to</u>
<u>Known Set:</u>
A

Dijkstra’s Algorithm: Example #1

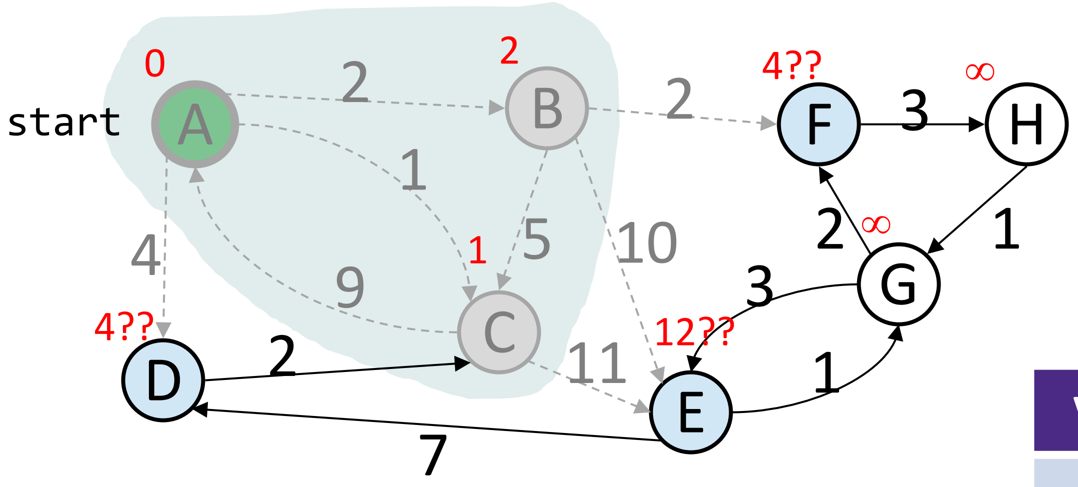


Order Added to Known Set:

A, C

Vertex	Known?	distTo	edgeTo
A	Y	0	/
B		≤ 2	A
C	Y	1	A
D		≤ 4	A
E		**≤ 12**	**C**
F		∞	
G		∞	
H		∞	

Dijkstra’s Algorithm: Example #1

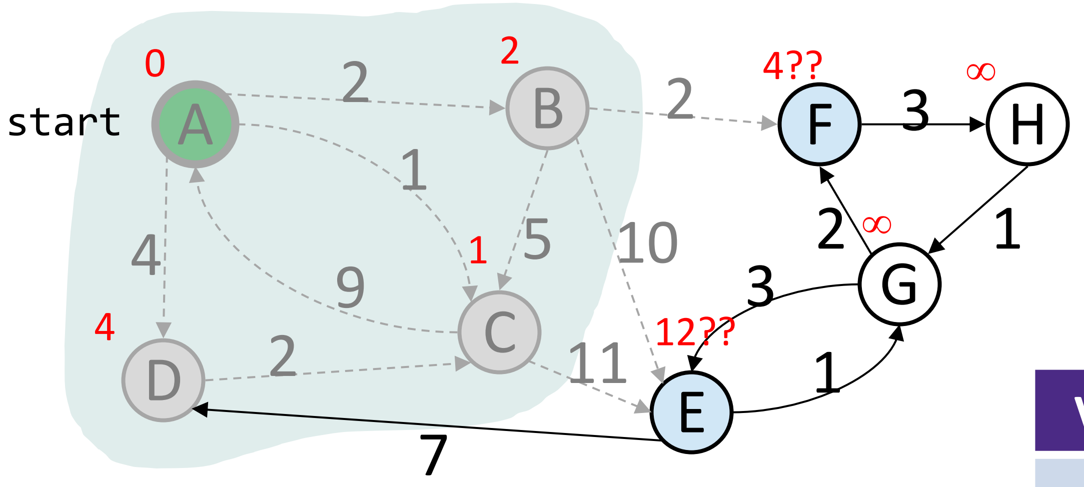


Vertex	Known?	distTo	edgeTo
A	Y	0	/
B	Y	2	A
C	Y	1	A
D		≤ 4	A
E		≤ 12	C
F		**≤ 4**	**B**
G		∞	
H		∞	

<u>Order Added to Known Set:</u>

A, C, B

Dijkstra's Algorithm: Example #1

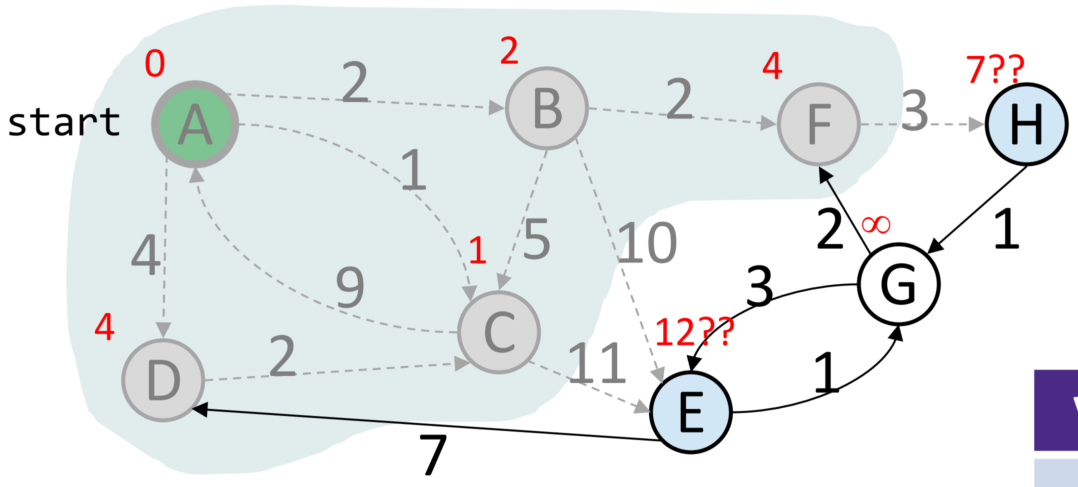


Vertex	Known?	distTo	edgeTo
A	Y	0	/
B	Y	2	A
C	Y	1	A
D	Y	4	A
E		≤ 12	C
F		≤ 4	B
G		∞	
H		∞	

<u>Order Added to Known Set:</u>

A, C, B, D

Dijkstra's Algorithm: Example #1

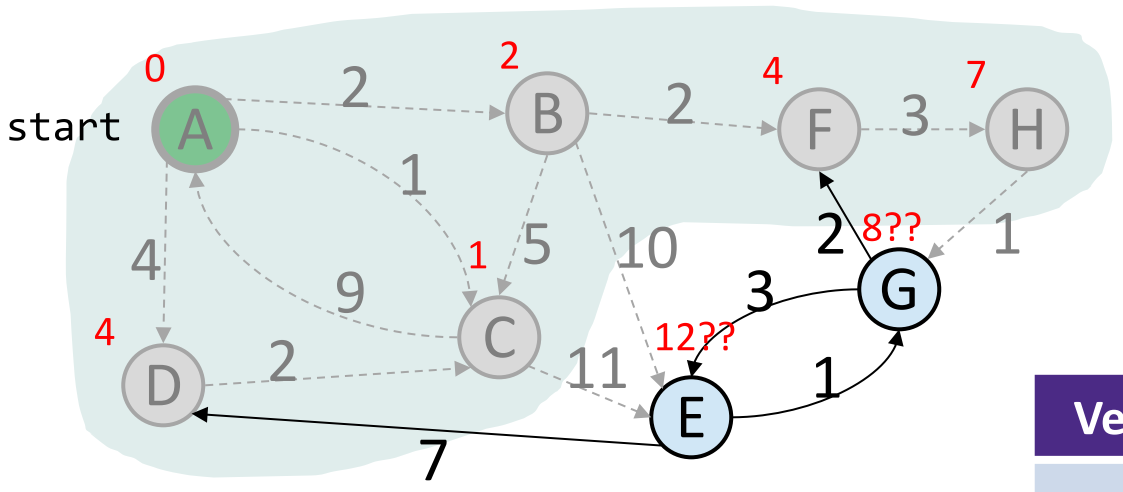


Vertex	Known?	distTo	edgeTo
A	Y	0	/
B	Y	2	A
C	Y	1	A
D	Y	4	A
E		≤ 12	C
F	Y	4	B
G		∞	
H		**≤ 7**	**F**

<u>Order Added to Known Set:</u>

A, C, B, D, F

Dijkstra’s Algorithm: Example #1

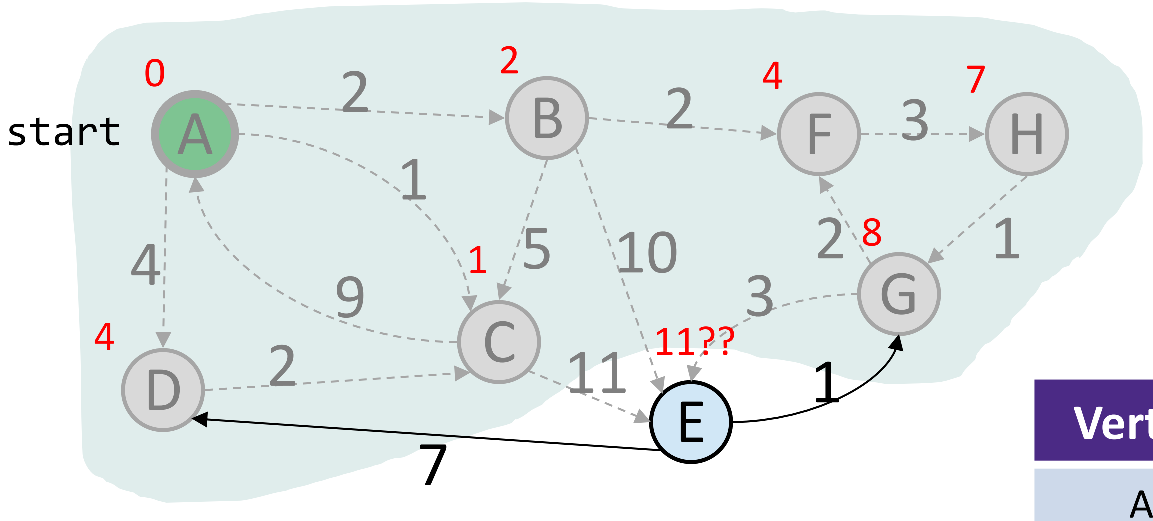


Vertex	Known?	distTo	edgeTo
A	Y	0	/
B	Y	2	A
C	Y	1	A
D	Y	4	A
E		≤ 12	C
F	Y	4	B
G		**≤ 8**	**H**
H	Y	7	F

<u>Order Added to Known Set:</u>

A, C, B, D, F, H

Dijkstra’s Algorithm: Example #1

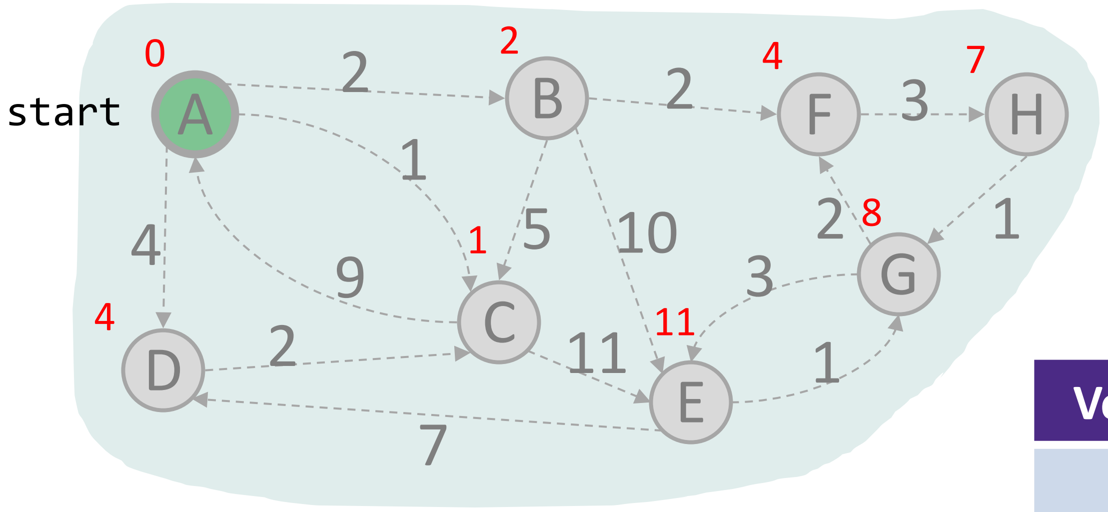


Vertex	Known?	distTo	edgeTo
A	Y	0	/
B	Y	2	A
C	Y	1	A
D	Y	4	A
E		**≤ 11**	**G**
F	Y	4	B
G	Y	8	H
H	Y	7	F

<u>Order Added to Known Set:</u>

A, C, B, D, F, H, G

Dijkstra's Algorithm: Example #1

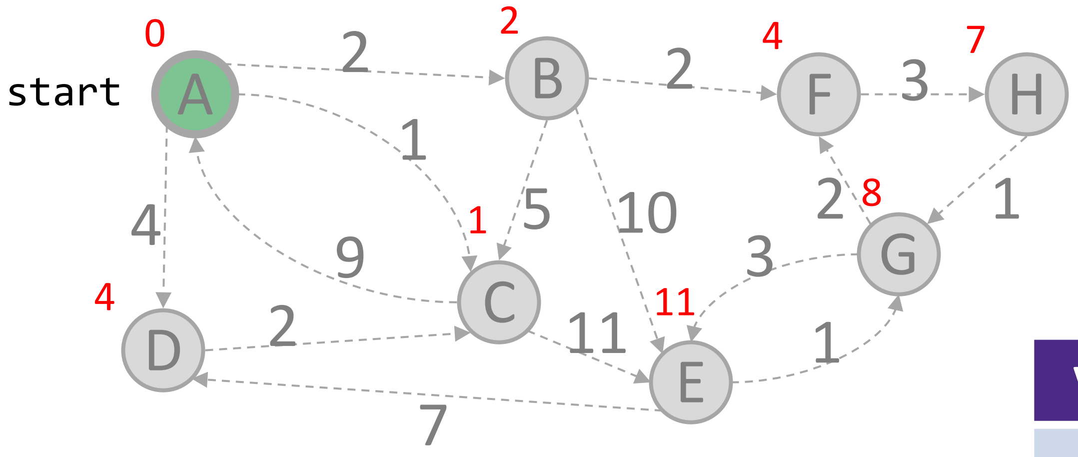


Order Added to Known Set:

A, C, B, D, F, H, G, E

Vertex	Known?	distTo	edgeTo
A	Y	0	/
B	Y	2	A
C	Y	1	A
D	Y	4	A
E	Y	11	G
F	Y	4	B
G	Y	8	H
H	Y	7	F

Dijkstra’s Algorithm: Interpreting the Results



Now that we’re done, how do we get the path from A to E?

Follow edgeTo backpointers!

distTo and edgeTo make up the **shortest path tree**

Vertex	Known?	distTo	edgeTo
A	Y	0	/
B	Y	2	A
C	Y	1	A
D	Y	4	A
E	Y	11	G
F	Y	4	B
G	Y	8	H
H	Y	7	F

Order Added to Known Set:

A, C, B, D, F, H, G, E

Review: Key Features

Once a vertex is marked known, its shortest path is known

- Can reconstruct path by following backpointers

While a vertex is not known, another shorter path might be found!

The “Order Added to Known Set” is unimportant

- A detail about how the algorithm works *(client doesn't care)*
- Not used by the algorithm *(implementation doesn't care)*
- It is sorted by path-distance; ties are resolved “somehow”

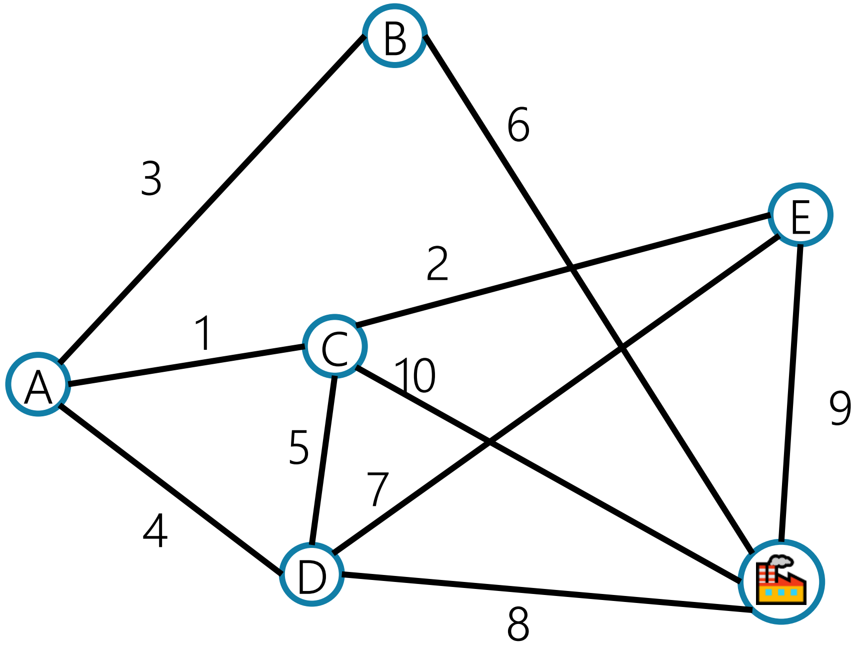
If we only need path to a specific vertex, can stop early once that vertex is known

- Because its shortest path cannot change!
- Return a partial **shortest path tree**

Minimum Spanning Trees

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 from the plant to every city.

MST Problem

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

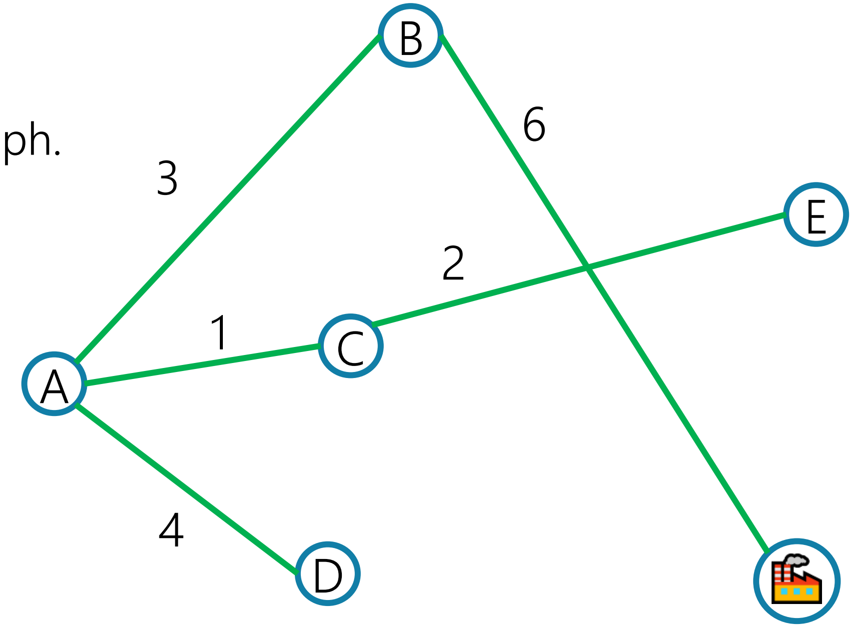
- Every vertex touches at least one of the edges. The edges "**span**" the graph.
- The graph on just those edges is **connected**.
- The minimum weight set of edges that meet those conditions.

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 connectiveness
- 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.

Interaction Pane Question:

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

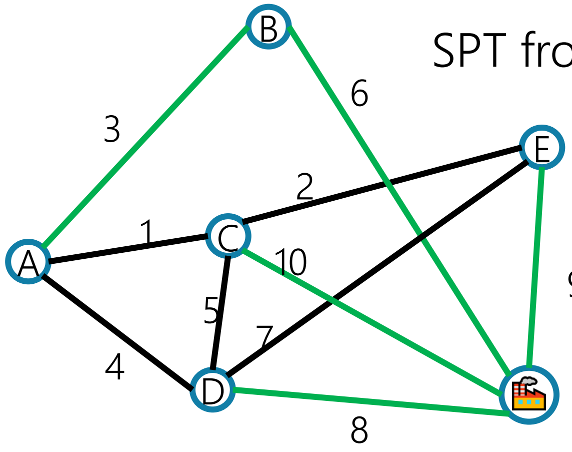
Shortest Path vs Minimum Spanning

Shortest Path Problem

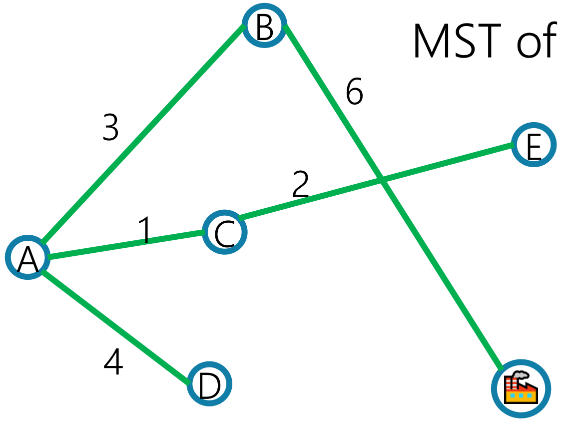
Given: a directed graph G and vertices s,t
Find: the shortest path from s to t.

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.



SPT from Factory



MST of the graph

<u>Shortest Path Tree</u>
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.

<u>Minimum Spanning Tree</u>
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

Finding an MST

Interaction Pane Question:

Which of these do you think are more likely to work?

- Thumbs up for vertex by vertex
- Thumbs down for edge by edge
- Clap for both

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!!

Prim’s Algorithm

In the Chat

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

```
Dijkstra(Graph G, Vertex source)
  initialize distances to ∞
  mark source as distance 0
  mark all vertices unprocessed
  while(there are unprocessed vertices){
     let u be the closest unprocessed vertex
     foreach(edge (u,v) leaving u){
        if(u.dist+weight(u,v) < v.dist){
           v.dist = u.dist+weight(u,v)
           v.predecessor = u
        }
     }
     mark u as processed
  }
```

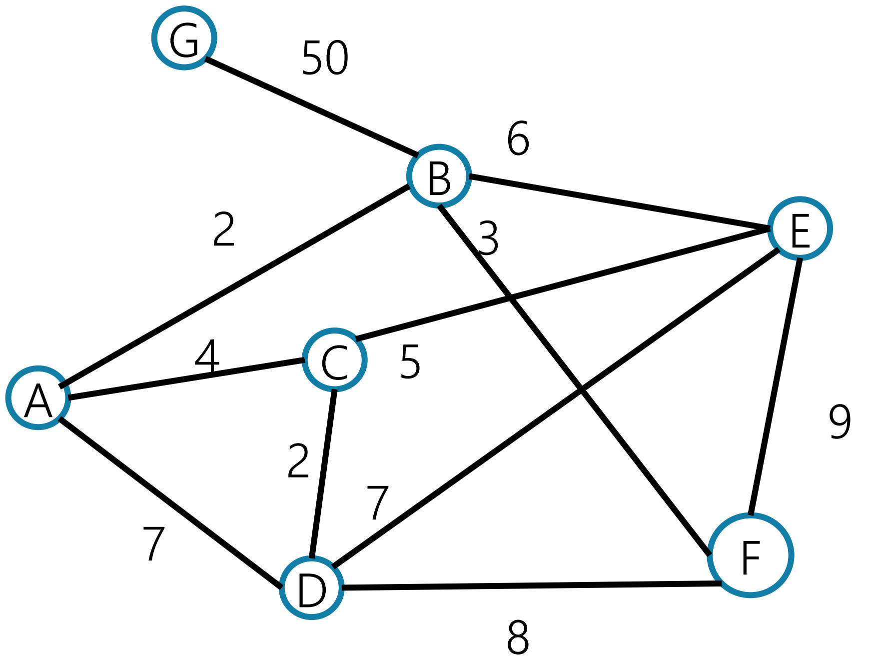
Algorithm idea:

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 processed

```
Prims(Graph G, Vertex source)
  initialize distances to ∞
  mark source as distance 0
  mark all vertices unprocessed
  while(there are unprocessed vertices){
     let u be the closest unprocessed vertex
     foreach(edge (u,v) leaving u){
        if(weight(u,v) < v.dist){
           v.dist = u.dist+weight(u,v)
           v.predecessor = u
        }
     }
     mark u as processed
  }
```

Try it Out

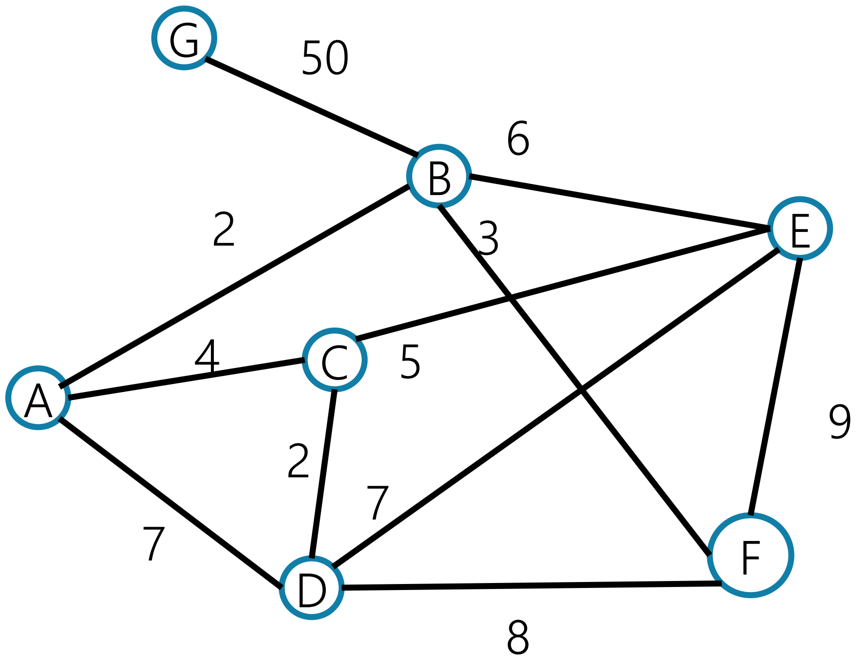
```
PrimMST(Graph G)
  initialize distances to ∞
  mark source as distance 0
  mark all vertices unprocessed
  foreach(edge (source, v) ) {
    v.dist = weight(source,v)
    v.bestEdge = (source,v)
  }
  while(there are unprocessed vertices){
    let u be the closest unprocessed vertex
    add u.bestEdge to spanning tree
    foreach(edge (u,v) leaving u){
      if(weight(u,v) < v.dist && v unprocessed ){
        v.dist = weight(u,v)
        v.bestEdge = (u,v)
      }
    }
    mark u as processed
  }
```



Vertex	Distance	Best Edge	Processed
A			
B			
C			
D			
E			
F			
G			

Try it Out

```
PrimMST(Graph G)
  initialize distances to ∞
  mark source as distance 0
  mark all vertices unprocessed
  foreach(edge (source, v) ) {
    v.dist = weight(source,v)
    v.bestEdge = (source,v)
  }
  while(there are unprocessed vertices){
    let u be the closest unprocessed vertex
    add u.bestEdge to spanning tree
    foreach(edge (u,v) leaving u){
      if(weight(u,v) < v.dist && v unprocessed ){
        v.dist = weight(u,v)
        v.bestEdge = (u,v)
      }
    }
    mark u as processed
  }
```



Vertex	Distance	Best Edge	Processed
A	-	X	✓
B	2	(A, B)	✓
C	4	(A, C)	✓
D	~~7~~ 2	~~(A, D)~~ (C, D)	✓
E	~~6~~ 5	~~(B, E)~~ (C, E)	✓
F	3	(B, F)	✓
G	50	(B, G)	✓

A different Approach

Prim's Algorithm started from a single vertex and reached more and more other vertices.

Prim's thinks vertex by vertex (add the closest vertex to the currently reachable set).

Prim's Algorithm Visualization

What if you think edge by edge instead?

Start from the lightest edge; add it if it connects new things to each other (don't add it if it would create a cycle)

This is Kruskal's Algorithm.

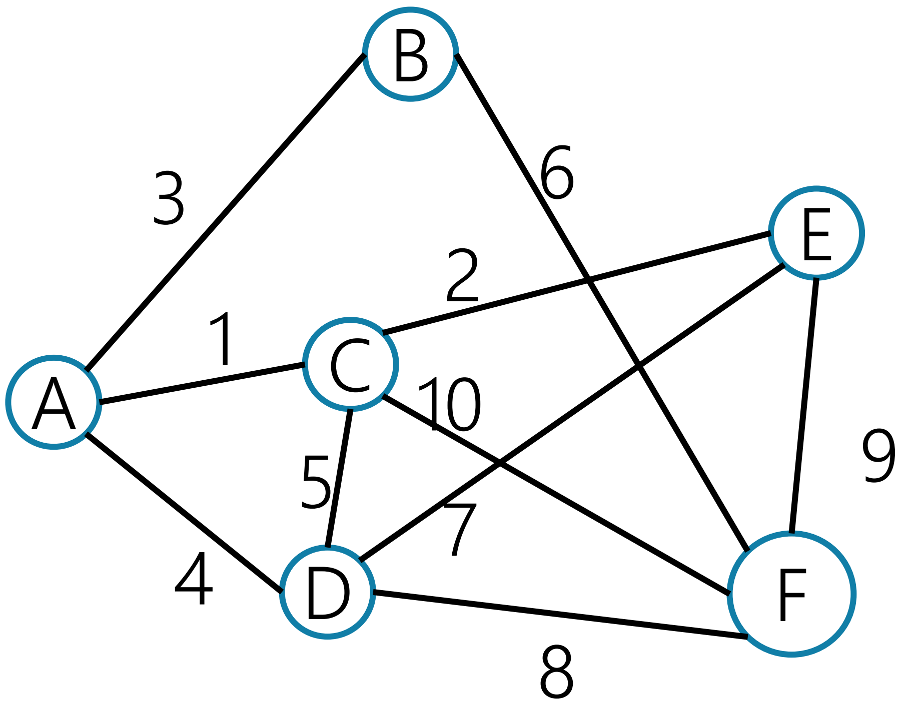
Kruskal's Algorithm Visualization

Kruskal’s Algorithm

```
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
      }
   }
```

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
         }
      }
```

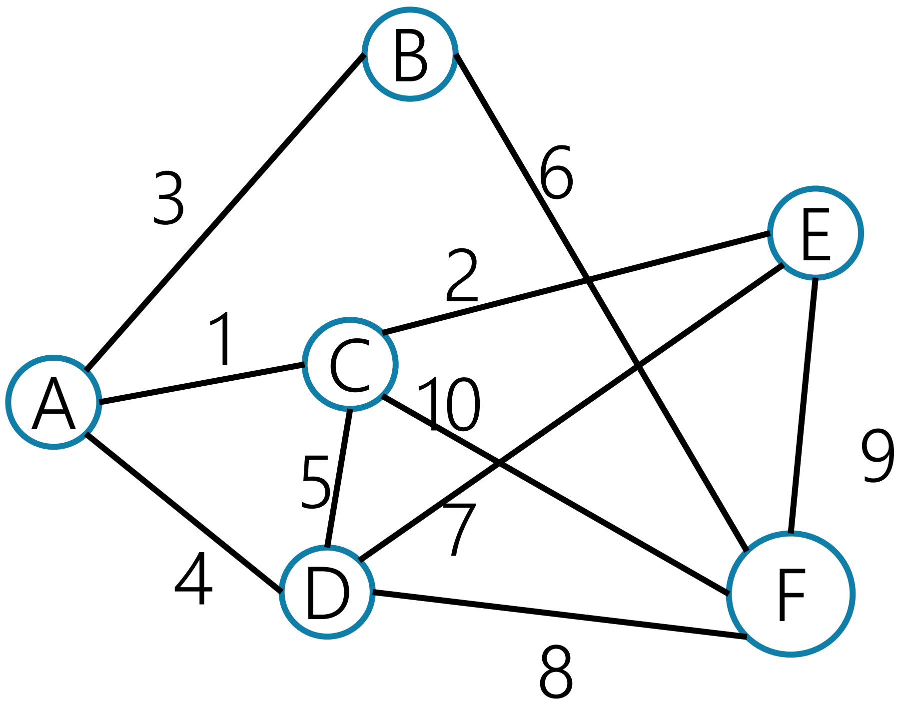


Edge	Include?	Reason
(A,C)		
(C,E)		
(A,B)		
(A,D)		
(C,D)		

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?