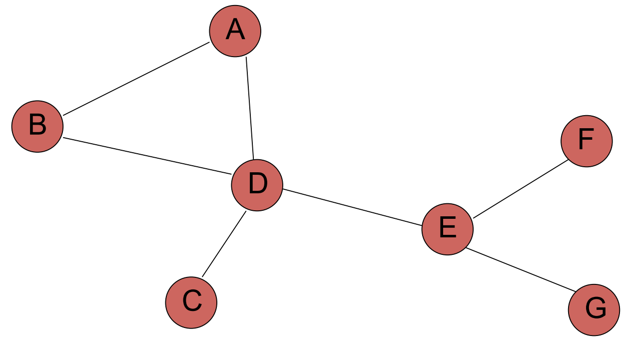
Minimum Spanning Trees

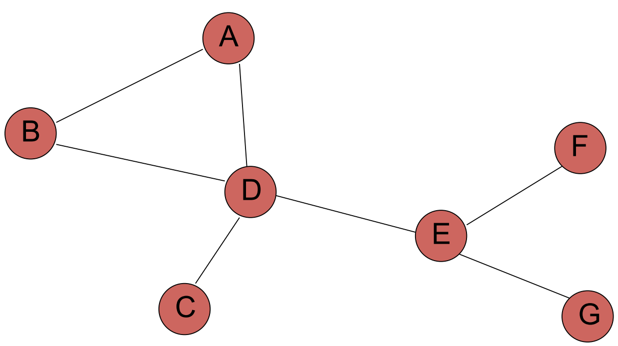
Graphs

A graph is a set of vertices V and a set of edges (u,v) ∈ E where u,v ∈ V



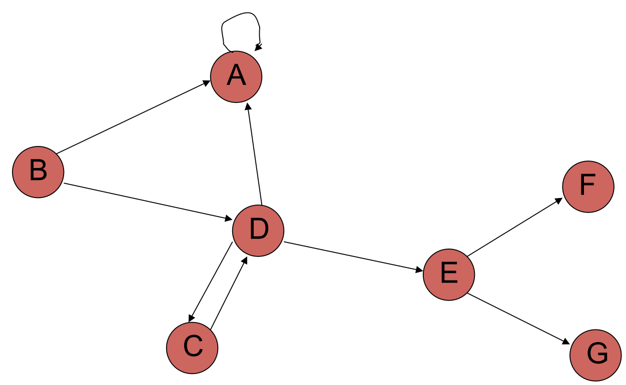
Different types of graphs

Undirected – edges do not have a direction



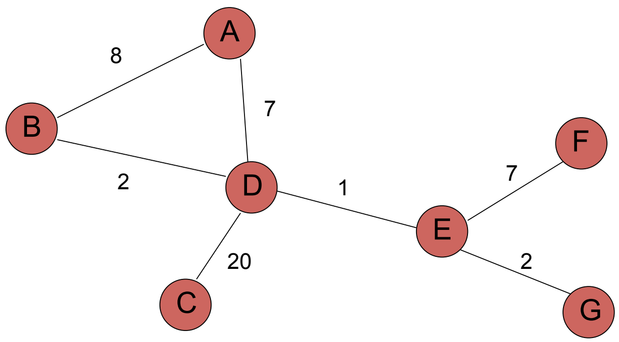
Different types of graphs

Directed – edges **do** have a direction



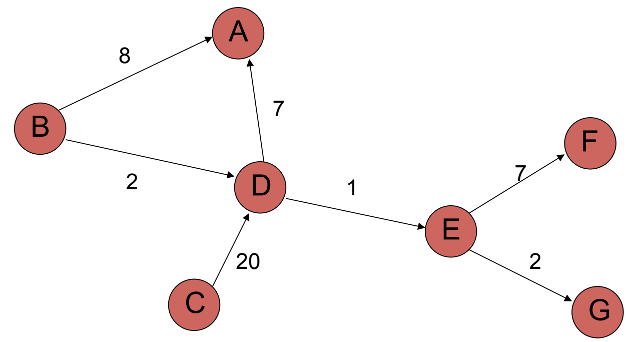
Different types of graphs

Weighted – edges have an associated weight



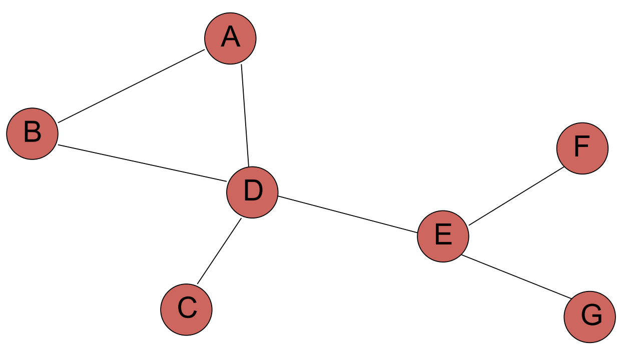
Different types of graphs

Weighted – edges have an associated weight



Terminology

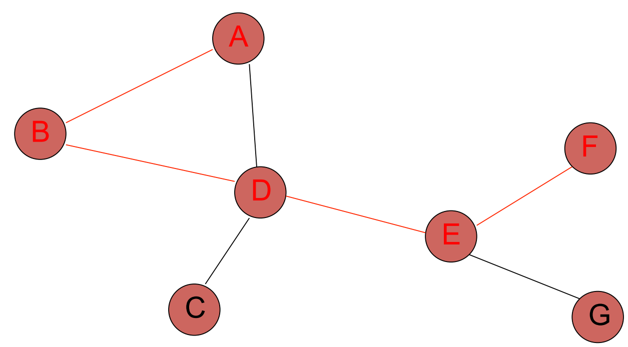
Path – A path is a list of vertices $p_1, p_2, \ldots p_k$ where there exists an edge $(p_i, p_{i+1}) \in E$



Terminology

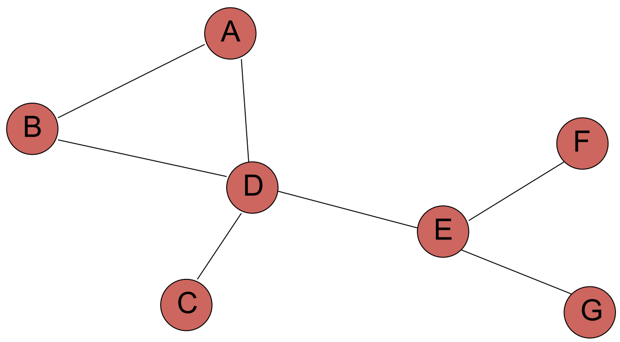
Path – A path is a list of vertices $p_1, p_2, \ldots p_k$ where there exists an edge $(p_i, p_{i+1}) \in E$

{A, B, D, E, F}



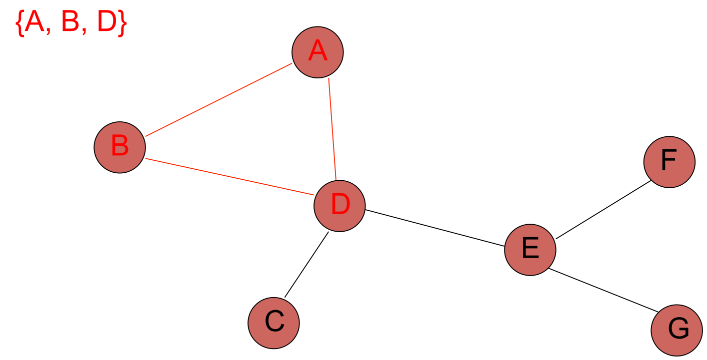
Terminology

Cycle – A subset of the edges that form a path such that the first and last node are the same



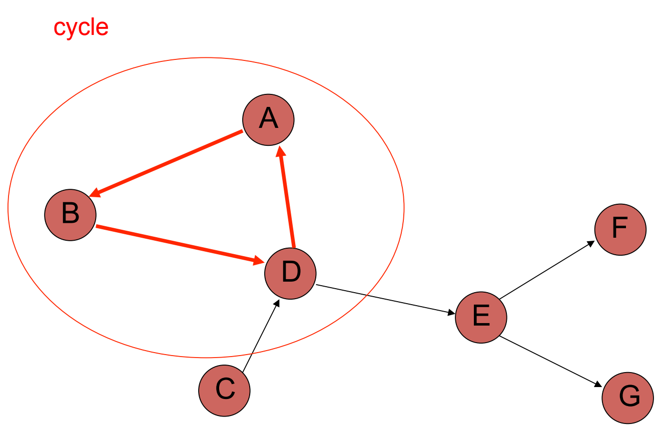
Terminology

Cycle – A subset of the edges that form a path such that the first and last node are the same



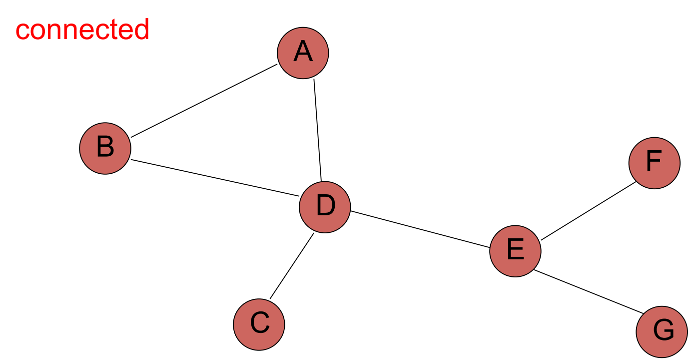
Terminology

Cycle – A path $p_1, p_2, \ldots p_k$ where $p_1 = p_k$



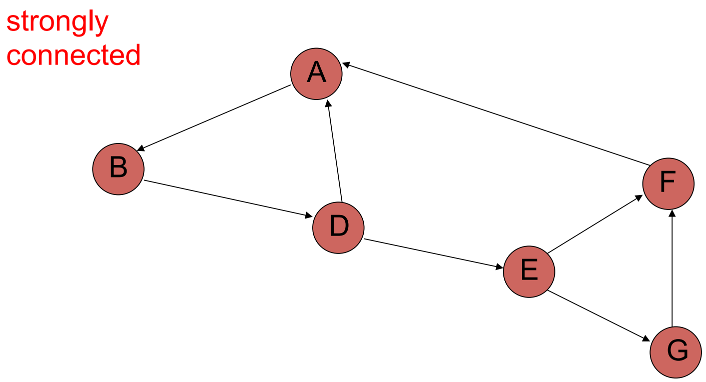
Terminology

Connected – every pair of vertices is connected by a path



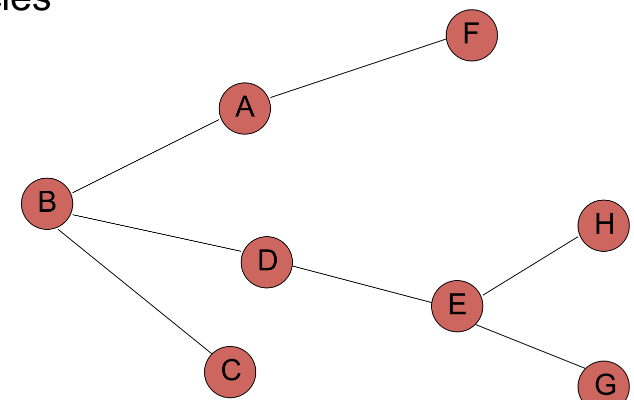
Terminology

Strongly connected (directed graphs) –
Every two vertices are reachable by a path



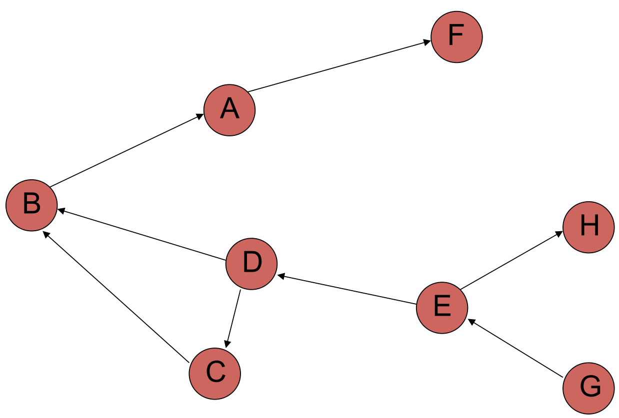
Different types of graphs

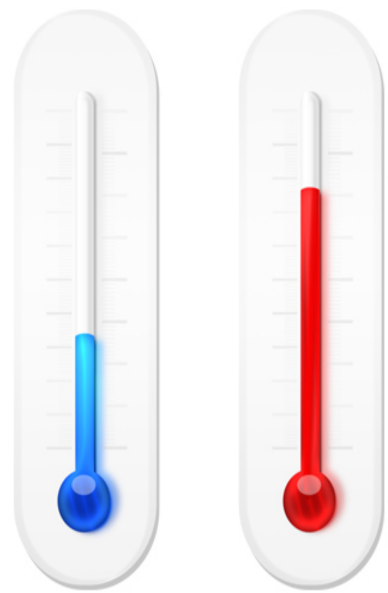
Tree – connected, undirected graph without any cycles



Different types of graphs

DAG – directed, acyclic graph





Road Repaving Problem

Minimum spanning trees

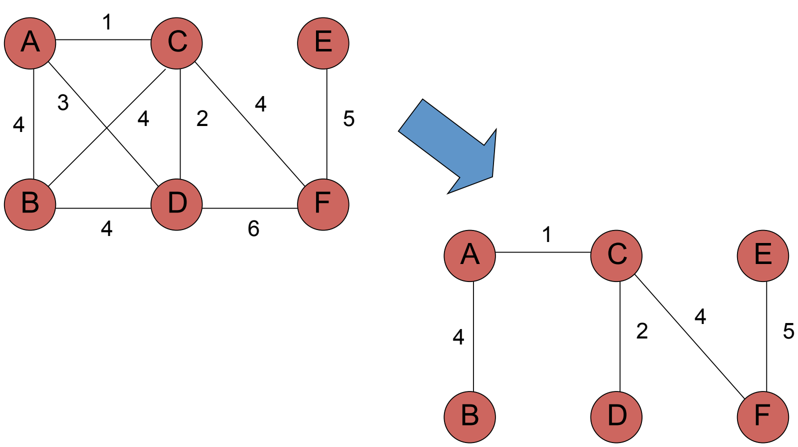
What is the lowest weight set of edges that connects all vertices of an undirected graph with positive weights

Input: An undirected, positive weight graph, G=(V,E)

Output: A tree T=(V,E') where E' $\subseteq$ E that minimizes

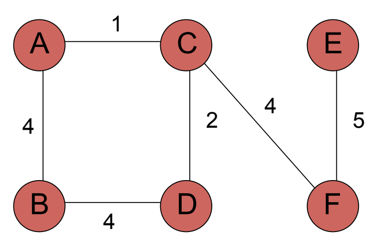
$$weight(T) = \sum_{e \in E'} w_e$$

MST example



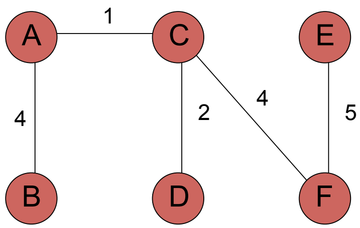
MSTs

Can an MST have a cycle?

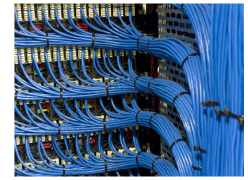


MSTs

How many edges will a MST have?



Applications?



Connectivity

- Networks (e.g. communications)
- Circuit design/wiring

Hub/spoke models (e.g. flights, transportation)

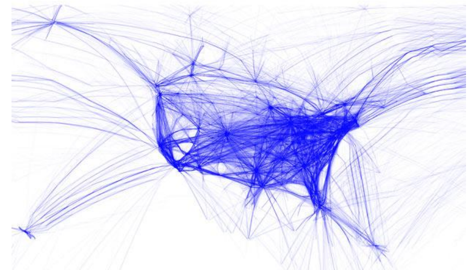
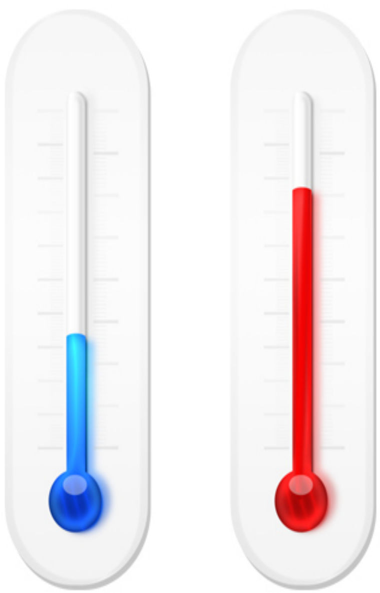
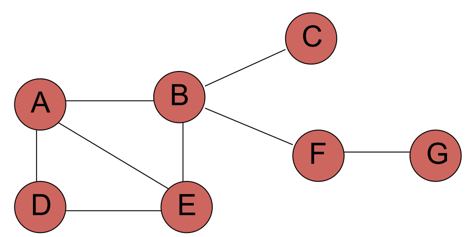


Photo Credit:
Aaron Koblin



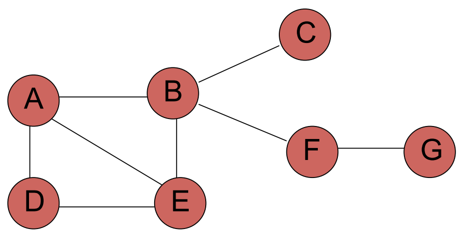
Worksheet:

- List all algorithms you can think of for finding *any* spanning tree for a graph (e.g., m.s.t. for graph with equal edge weights).



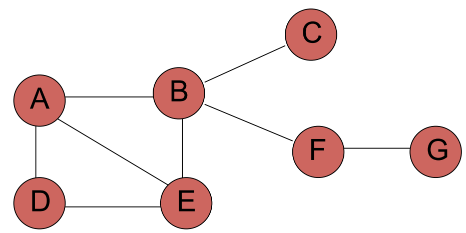
```
BFS(G, s)
for each v ∈ V
        dist[v] = ∞
dist[s] = 0
Enqueue(Q, s)
while !Empty(Q)
        u ← Dequeue(Q)
        Visit(u)
        for each edge (u, v) ∈ E
                if dist[v] = ∞
                        Enqueue(Q, v)
                        dist[v] ← dist[u] + 1
```

```
TreeBFS(T)
Enqueue(Q, Root(T))
while !Empty(Q)
        v ← Dequeue(Q)
        Visit(v)
        for all c ∈ Children(v)
                Enqueue(Q, c)
```



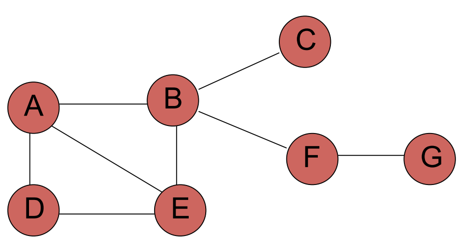

```
BFS(G, s)
for each v ∈ V
        dist[v] = ∞
dist[s] = 0
Enqueue(Q, s)
while !Empty(Q)
        u ← Dequeue(Q)
        Visit(u)
        for each edge (u, v) ∈ E
                if dist[v] = ∞
                        Enqueue(Q, v)
                        dist[v] ← dist[u] + 1
```

set all nodes as unseen



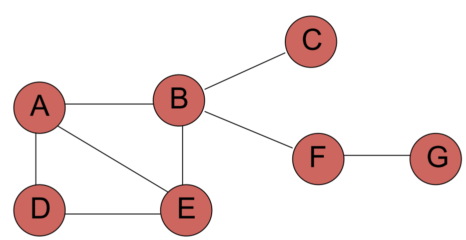

```
BFS(G, s)
for each v ∈ V
        dist[v] = ∞
dist[s] = 0
Enqueue(Q, s)
while !Empty(Q)
        u ← Dequeue(Q)
        Visit(u)
        for each edge (u, v) ∈ E
                if dist[v] = ∞
                        Enqueue(Q, v)
                        dist[v] ← dist[u] + 1
```

check if the node has been seen

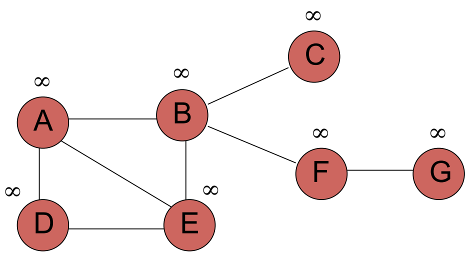



```
BFS(G, s)
for each v ∈ V
        dist[v] = ∞
dist[s] = 0
Enqueue(Q, s)
while !Empty(Q)
        u ← Dequeue(Q)
        Visit(u)
        for each edge (u, v) ∈ E
                if dist[v] = ∞
                        Enqueue(Q, v)
                        dist[v] ← dist[u] + 1
```

set the node as seen and record distance

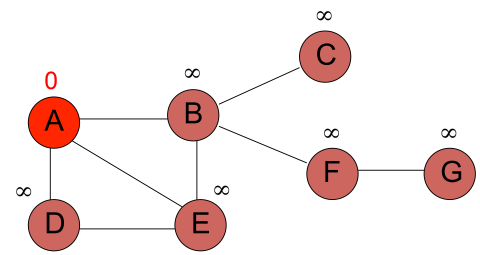


```
BFS(G, s)
for each v ∈ V
        dist[v] = ∞
dist[s] = 0
ENQUEUE(Q, s)
while !EMPTY(Q)
        u ← DEQUEUE(Q)
        VISIT(U)
        for each edge (u, v) ∈ E
                if dist[v] = ∞
                        ENQUEUE(Q, v)
                        dist[v] ← dist[u] + 1
```



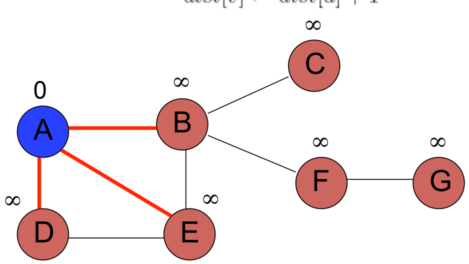

```
BFS(G, s)
for each v ∈ V
        dist[v] = ∞
dist[s] = 0
ENQUEUE(Q, s)
while !EMPTY(Q)
        u ← DEQUEUE(Q)
        VISIT(U)
        for each edge (u, v) ∈ E
                if dist[v] = ∞
                        ENQUEUE(Q, v)
                        dist[v] ← dist[u] + 1
```

Q: A



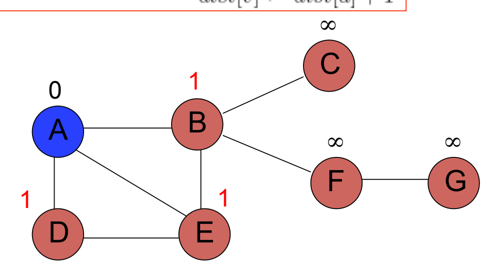

```
BFS(G, s)
for each v ∈ V
        dist[v] = ∞
dist[s] = 0
ENQUEUE(Q, s)
while !EMPTY(Q)
        u ← DEQUEUE(Q)
        VISIT(U)
        for each edge (u, v) ∈ E
                if dist[v] = ∞
                        ENQUEUE(Q, v)
                        dist[v] ← dist[u] + 1
```

Q:



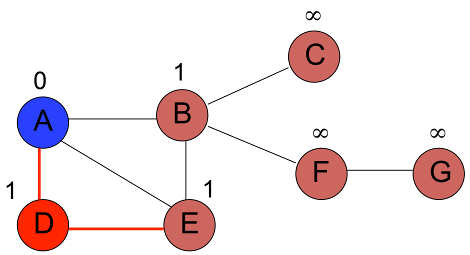

```
BFS(G, s)
for each v ∈ V
        dist[v] = ∞
dist[s] = 0
ENQUEUE(Q, s)
while !EMPTY(Q)
        u ← DEQUEUE(Q)
        VISIT(U)
        for each edge (u, v) ∈ E
                if dist[v] = ∞
                        ENQUEUE(Q, v)
                        dist[v] ← dist[u] + 1
```

Q: D, E, B



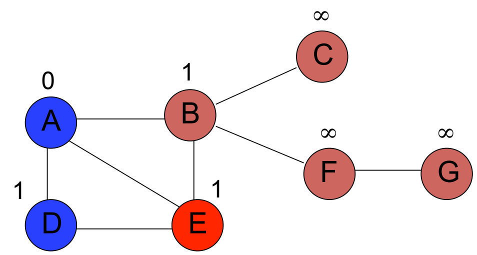
```
BFS(G, s)
for each v ∈ V
        dist[v] = ∞
dist[s] = 0
ENQUEUE(Q, s)
while !EMPTY(Q)
        u ← DEQUEUE(Q)
        VISIT(U)
        for each edge (u, v) ∈ E
                if dist[v] = ∞
                        ENQUEUE(Q, v)
                        dist[v] ← dist[u] + 1
```

Q: E, B



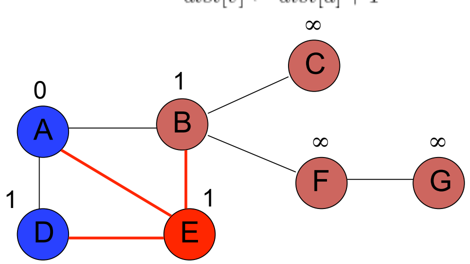

```
BFS(G, s)
for each v ∈ V
        dist[v] = ∞
dist[s] = 0
ENQUEUE(Q, s)
while !EMPTY(Q)
        u ← DEQUEUE(Q)
        VISIT(U)
        for each edge (u, v) ∈ E
                if dist[v] = ∞
                        ENQUEUE(Q, v)
                        dist[v] ← dist[u] + 1
```

Q: B



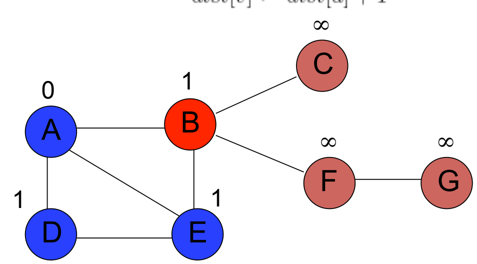

```
BFS(G, s)
for each v ∈ V
        dist[v] = ∞
dist[s] = 0
ENQUEUE(Q, s)
while !EMPTY(Q)
        u ← DEQUEUE(Q)
        VISIT(U)
        for each edge (u, v) ∈ E
                if dist[v] = ∞
                        ENQUEUE(Q, v)
                        dist[v] ← dist[u] + 1
```

Q: B



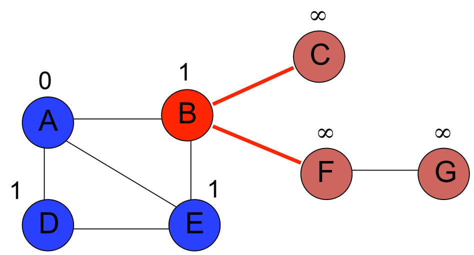

```
BFS(G, s)
for each v ∈ V
        dist[v] = ∞
dist[s] = 0
ENQUEUE(Q, s)
while !EMPTY(Q)
        u ← DEQUEUE(Q)
        VISIT(U)
        for each edge (u, v) ∈ E
                if dist[v] = ∞
                        ENQUEUE(Q, v)
                        dist[v] ← dist[u] + 1
```

Q:



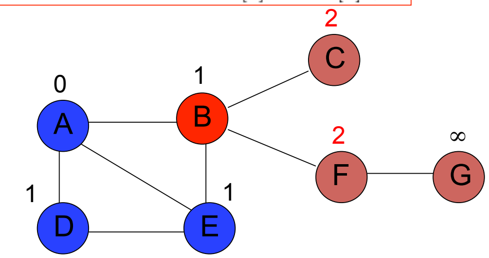
```
BFS(G, s)
for each v ∈ V
        dist[v] = ∞
dist[s] = 0
ENQUEUE(Q, s)
while !EMPTY(Q)
        u ← DEQUEUE(Q)
        VISIT(U)
        for each edge (u, v) ∈ E
                if dist[v] = ∞
                        ENQUEUE(Q, v)
                        dist[v] ← dist[u] + 1
```

Q:

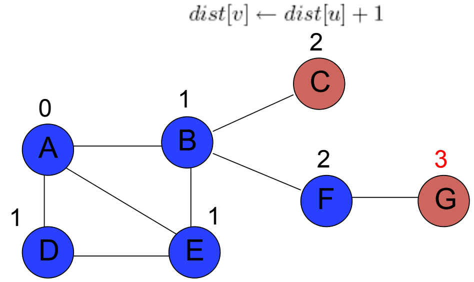



```
BFS(G, s)
for each v ∈ V
        dist[v] = ∞
dist[s] = 0
ENQUEUE(Q, s)
while !EMPTY(Q)
        u ← DEQUEUE(Q)
        VISIT(U)
        for each edge (u, v) ∈ E
                if dist[v] = ∞
                        ENQUEUE(Q, v)
                        dist[v] ← dist[u] + 1
```

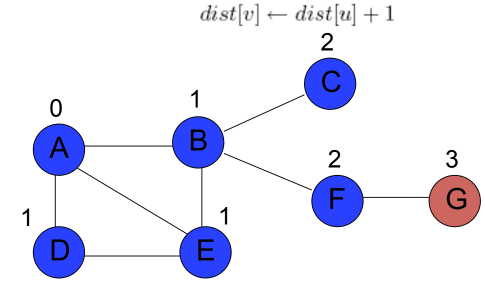
Q: F, C




```
BFS(G, s)
for each v ∈ V
        dist[v] = ∞
dist[s] = 0
ENQUEUE(Q, s)
while !EMPTY(Q)
        u ← DEQUEUE(Q)
        VISIT(U)
        for each edge (u, v) ∈ E
                if dist[v] = ∞
                        ENQUEUE(Q, v)
                        dist[v] ← dist[u] + 1
```

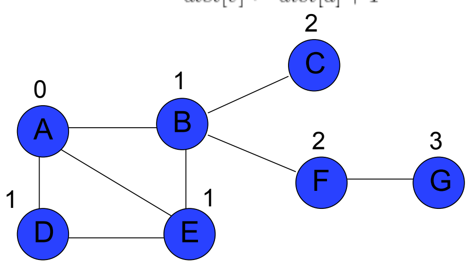



```
BFS(G, s)
for each v ∈ V
        dist[v] = ∞
dist[s] = 0
ENQUEUE(Q, s)
while !EMPTY(Q)
        u ← DEQUEUE(Q)
        VISIT(U)
        for each edge (u, v) ∈ E
                if dist[v] = ∞
                        ENQUEUE(Q, v)
                        dist[v] ← dist[u] + 1
```



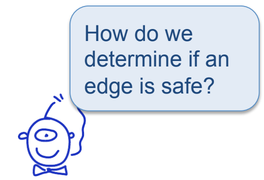
BFS(G, s)

```
for each v ∈ V
        dist[v] = ∞
dist[s] = 0
Enqueue(Q, s)
while !Empty(Q)
        u ← Dequeue(Q)
        Visit(u)
        for each edge (u, v) ∈ E
                if dist[v] = ∞
                        Enqueue(Q, v)
                        dist[v] ← dist[u] + 1
```



The "Generic" Greedy MST Algorithm

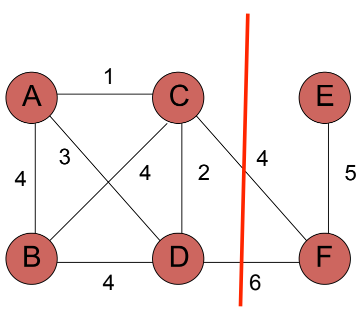
```
MST(Graph G):
  T = {}  # The tree we're growing!
  while |T| < n-1  # n is num vertices
     Find a safe edge e in G – T
     T = T + e
  return T
```



Cuts

A cut is a partitioning of the vertices into two sets S and V-S

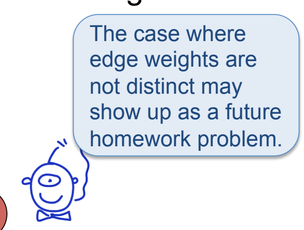
An edge "crosses" the cut if it connects a vertex u∈V and v∈V-S

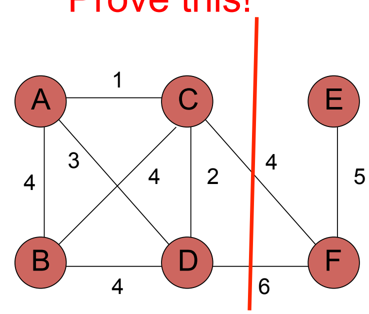


Minimum cut property

Given a partition S, let the light edge *e* be the minimum cost edge that **crosses** the partition. *Every* minimum spanning tree contains the light edge *e*.

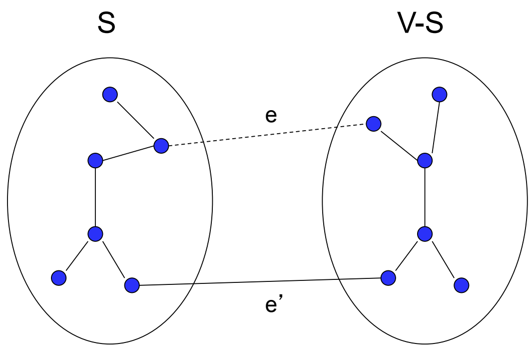
Worksheet: Prove this!





Minimum cut property

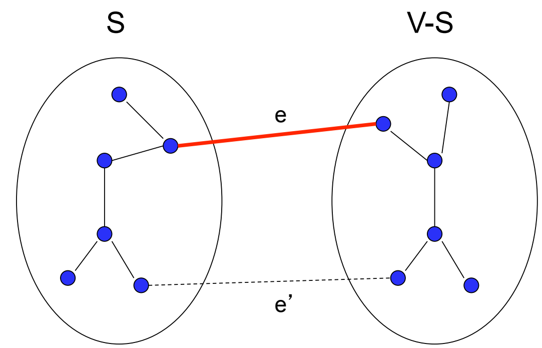
Given a partion S, let edge *e* be the minimum cost edge that **crosses** the partition. *Every* minimum spanning tree contains edge *e*.



Consider an MST with edge e' that is not the minimum edge

Minimum cut property

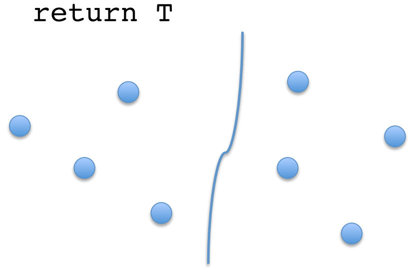
Given a partion S, let edge *e* be the minimum cost edge that **crosses** the partition. *Every* minimum spanning tree contains edge *e*.

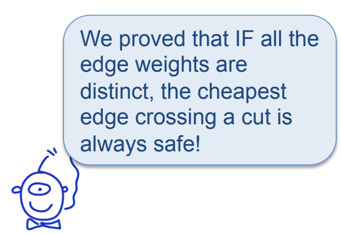


Using e instead of e', still connects the graph, but produces a tree with smaller weights

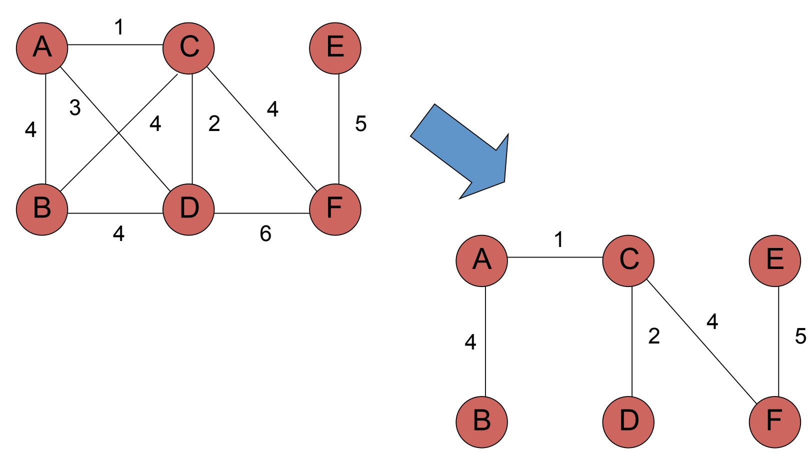
The "Generic" Greedy MST Algorithm

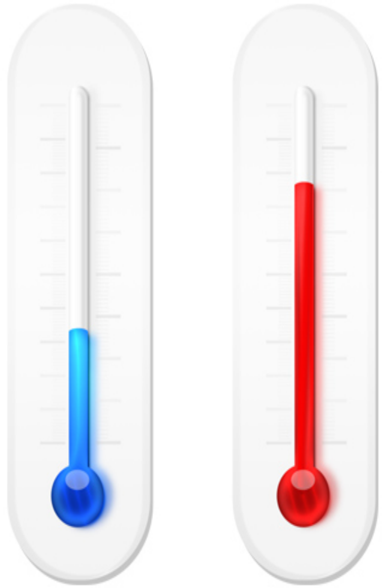
```
MST(Graph G):
  T = {}  # The tree we're growing!
  while |T| < n-1  # n is num vertices
     Find a safe edge e in G – T
     T = T + e
  return T
```





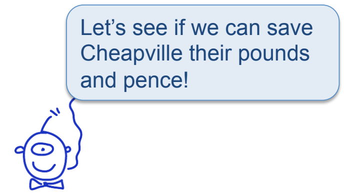
Algorithm ideas?





Kruskal's Algorithm (The idea)

- Start with each vertex being its own component
- Repeated merge components into on by choosing the light edge that connects them
  - Sort all edge in monotonically increasing order by weight
  - Use a disjoint-set data structure to determine whether an edge connects vertices in difference components



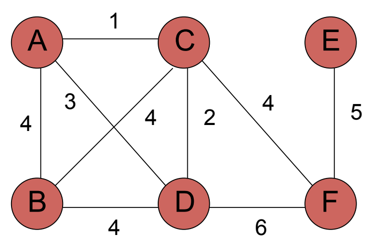
Kruskal's algorithm

Given a partition S, let edge *e* be the minimum cost edge that **crosses** the partition. *Every* minimum spanning tree contains edge *e*.

```
KRUSKAL(G)
for all v ∈ V
        MAKESET(v)
T ← {}
sort the edges of E by weight
for all edges (u, v) ∈ E in increasing order of weight
        if FIND-SET(u) ≠ FIND-SET(v)
                add edge to T
                UNION(FIND-SET(u),FIND-SET(v))
```

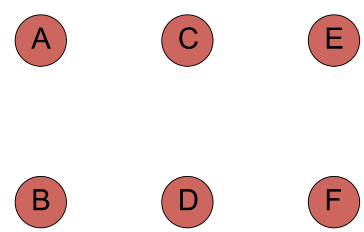
Kruskal's algorithm

Add smallest edge that connects two sets not already connected



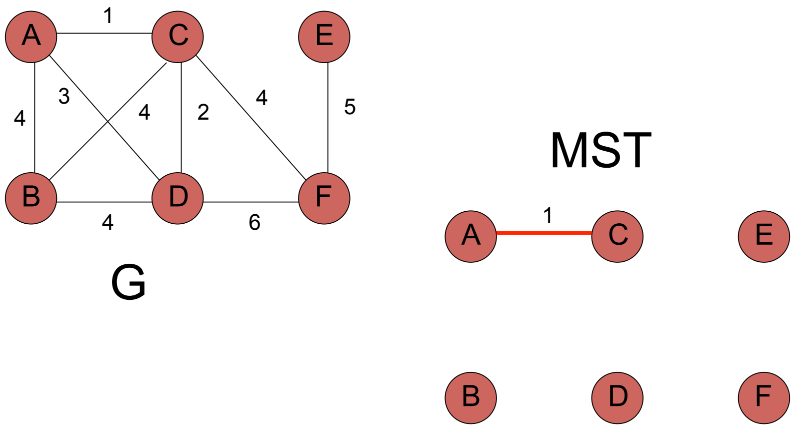
G

MST



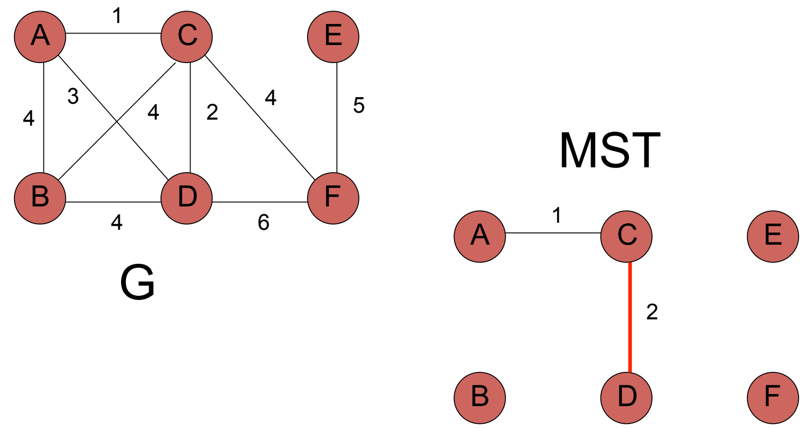
Kruskal’s algorithm

Add smallest edge that connects
two sets not already connected



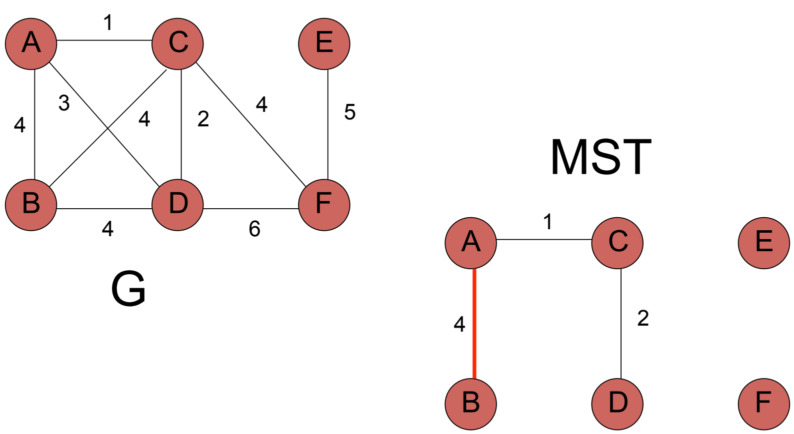
Kruskal’s algorithm

Add smallest edge that connects
two sets not already connected



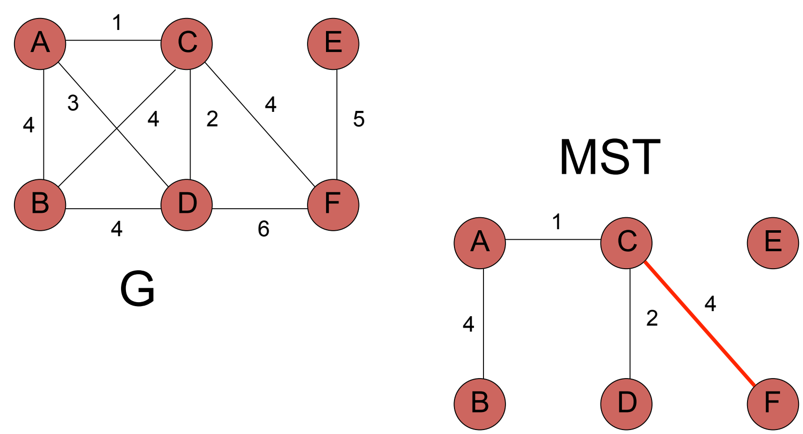
Kruskal’s algorithm

Add smallest edge that connects
two sets not already connected



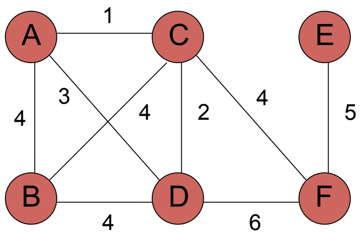
Kruskal’s algorithm

Add smallest edge that connects
two sets not already connected



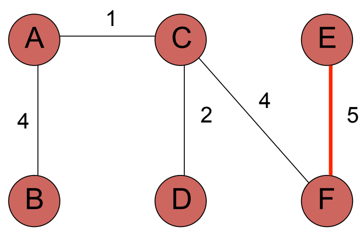
Kruskal's algorithm

Add smallest edge that connects two sets not already connected



G

MST



Correctness of Kruskal's

Never adds an edge that connects already connected vertices

Always adds lowest cost edge to connect two sets. By min cut property, that edge must be part of the MST

```
KRUSKAL(G)
for all v ∈ V
        MAKESET(v)
T ← {}
sort the edges of E by weight
for all edges (u, v) ∈ E in increasing order of weight
        if FIND-SET(u) ≠ FIND-SET(v)
                add edge to T
                UNION(FIND-SET(u),FIND-SET(v))
```

Running time of Kruskal's

```
KRUSKAL(G)
for all v ∈ V
        MAKESET(v)
T ← {}
sort the edges of E by weight
for all edges (u, v) ∈ E in increasing order of weight
        if FIND-SET(u) ≠ FIND-SET(v)
                add edge to T
                UNION(FIND-SET(u),FIND-SET(v))
```

Running time of Kruskal's

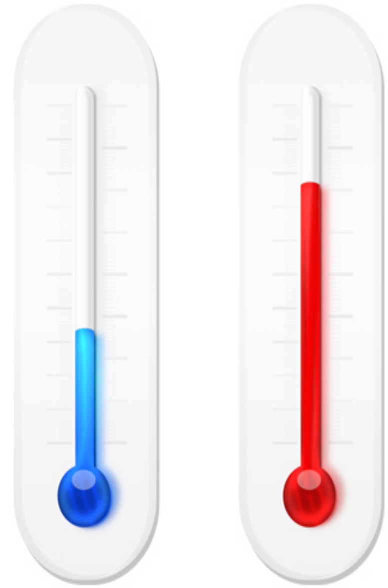
```
KRUSKAL(G)
for all v ∈ V                                              |V| calls to MakeSet
        MAKESET(v)
T ← {}
sort the edges of E by weight                              O(|E| log |E|)
for all edges (u, v) ∈ E in increasing order of weight
        if FIND-SET(u) ≠ FIND-SET(v)                       2 |E| calls to FindSet
                add edge to T
                UNION(FIND-SET(u),FIND-SET(v))             |V| calls to Union
```

Running time of Kruskal's

Disjoint set data structure

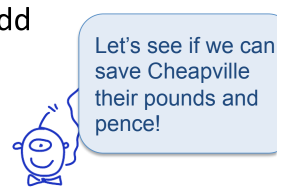
O(|E| log |E|) +

	MakeSet	FindSet |E| calls	Union |V| calls	Total
Linked lists	|V|	O(|V| |E|)	|V|	O(|V||E| + |E| log |E|) O(|V| |E|)
Linked lists + heuristics	|V|	O(|E| log |V|)	|V|	O(|E| log |V|+ |E| log |E|) O(|E| log |E|)



Prim's Algorithm (The Idea)

- Incrementally build tree
  - Pick any root
  - At each step, find light edge that crosses the cut T, G-T
- Implemented using a priority queue
  - Priorities represent costs connect to T
  - Extract-Min chooses the edge to add
  - Decrease-Key updates priorities of crossing edges



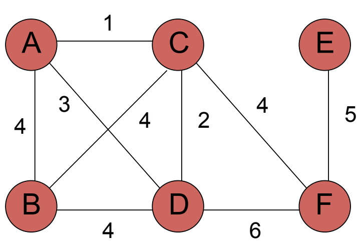
Prim's algorithm

```
Prim(G, r)
for all v ∈ V
        key[v] ← ∞
        prev[v] ← null
key[r] ← 0
H ← MakeHeap(key)
while !Empty(H)
        u ← Extract-Min(H)
        visited[u] ← true
        for each edge (u, v) ∈ E
                if !visited[v] and w(u, v) < key(v)
                        Decrease-Key(v, w(u, v))
                        prev[v] ← u
```

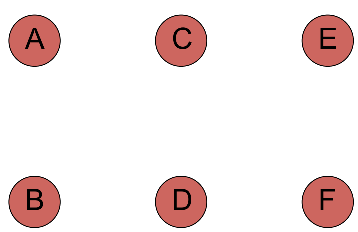
Prim's algorithm

```
Prim(G, r)
for all v ∈ V
        key[v] ← ∞
        prev[v] ← null
key[r] ← 0
H ← MakeHeap(key)
while !Empty(H)
        u ← Extract-Min(H)
        visited[u] ← true
        for each edge (u, v) ∈ E
                if !visited[v] and w(u, v) < key(v)
                        Decrease-Key(v, w(u, v))
                        prev[v] ← u
```

```
while !Empty(H)
        u ← Extract-Min(H)
        visited[u] ← true
        for each edge (u, v) ∈ E
                if !visited[v] and w(u, v) < key(v)
                        Decrease-Key(v, w(u, v))
                        prev[v] ← u
```



MST

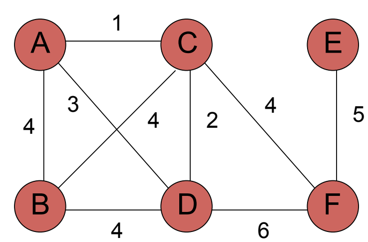


Prim's algorithm

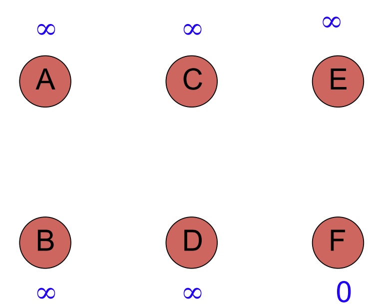
Start at some root node and build out the MST by adding the lowest weighted edge at the frontier

```
Prim(G, r)
for all v ∈ V
        key[v] ← ∞
        prev[v] ← null
key[r] ← 0
H ← MakeHeap(key)
while !Empty(H)
        u ← Extract-Min(H)
        visited[u] ← true
        for each edge (u, v) ∈ E
                if !visited[v] and w(u, v) < key(v)
                        Decrease-Key(v, w(u, v))
                        prev[v] ← u
```

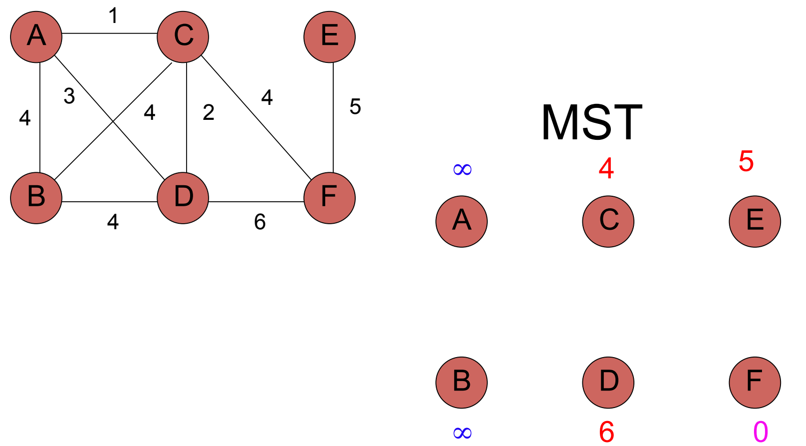
```
while !Empty(H)
        u ← Extract-Min(H)
        visited[u] ← true
        for each edge (u, v) ∈ E
                if !visited[v] and w(u, v) < key(v)
                        Decrease-Key(v, w(u, v))
                        prev[v] ← u
```



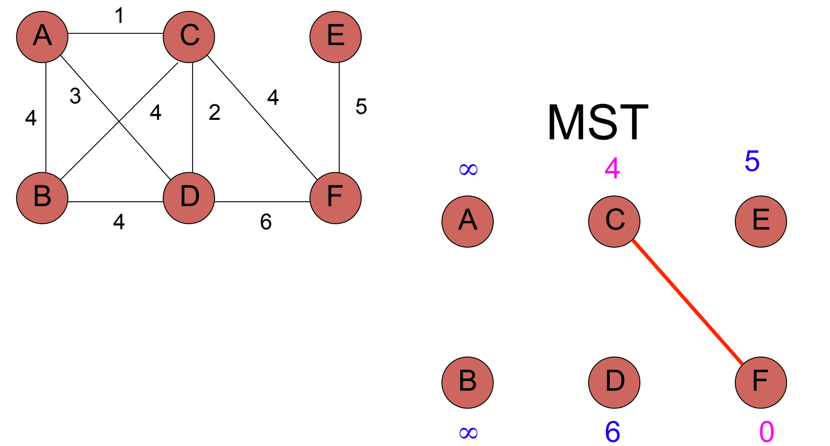
MST



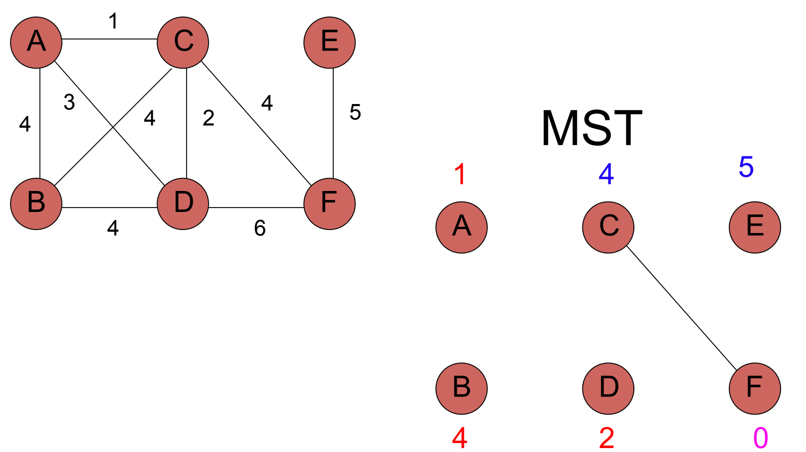
```
while !Empty(H)
        u ← Extract-Min(H)
        visited[u] ← true
        for each edge (u, v) ∈ E
                if !visited[v] and w(u, v) < key(v)
                        Decrease-Key(v, w(u, v))
                        prev[v] ← u
```



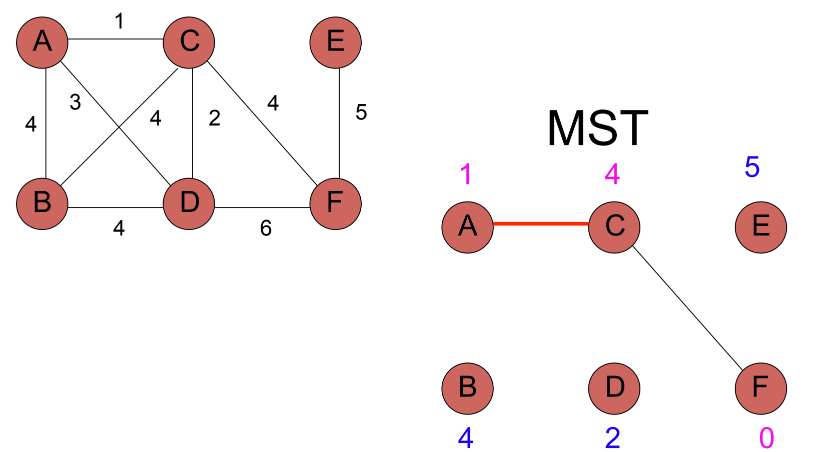

```
while !Empty(H)
        u ← Extract-Min(H)
        visited[u] ← true
        for each edge (u, v) ∈ E
                if !visited[v] and w(u, v) < key(v)
                        Decrease-Key(v, w(u, v))
                        prev[v] ← u
```



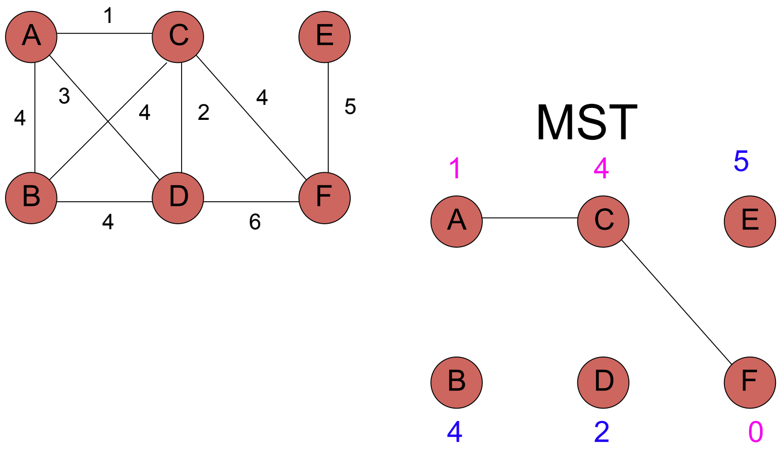

```
while !Empty(H)
        u ← Extract-Min(H)
        visited[u] ← true
        for each edge (u, v) ∈ E
                if !visited[v] and w(u, v) < key(v)
                        Decrease-Key(v, w(u, v))
                        prev[v] ← u
```



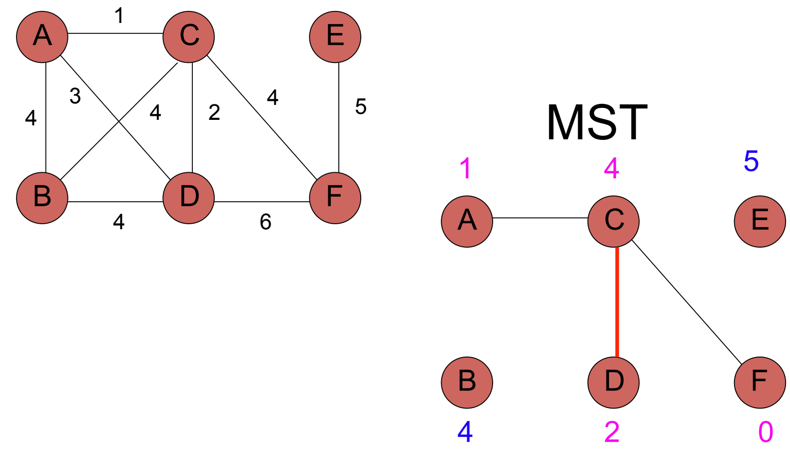

```
while !Empty(H)
        u ← Extract-Min(H)
        visited[u] ← true
        for each edge (u, v) ∈ E
                if !visited[v] and w(u, v) < key(v)
                        Decrease-Key(v, w(u, v))
                        prev[v] ← u
```



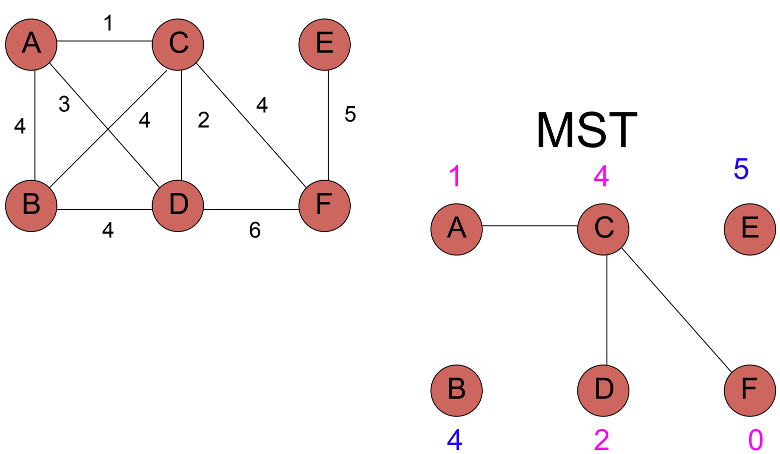
```
while !Empty(H)
    u ← Extract-Min(H)
    visited[u] ← true
    for each edge (u, v) ∈ E
        if !visited[v] and w(u, v) < key(v)
            Decrease-Key(v, w(u, v))
            prev[v] ← u
```



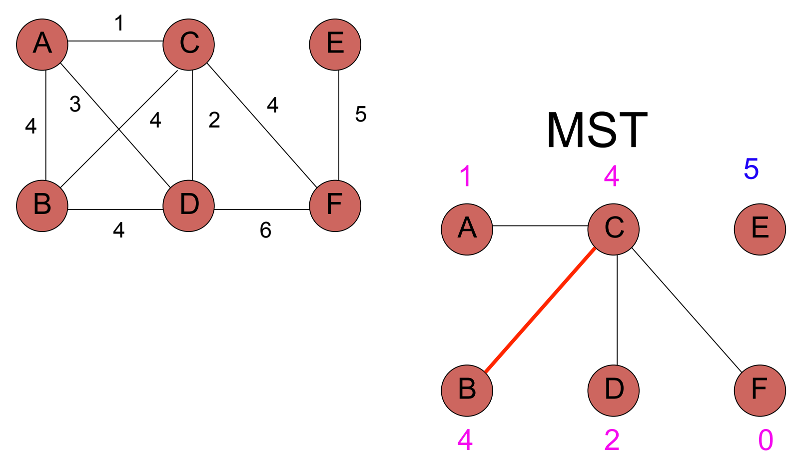

```
while !Empty(H)
    u ← Extract-Min(H)
    visited[u] ← true
    for each edge (u, v) ∈ E
        if !visited[v] and w(u, v) < key(v)
            Decrease-Key(v, w(u, v))
            prev[v] ← u
```



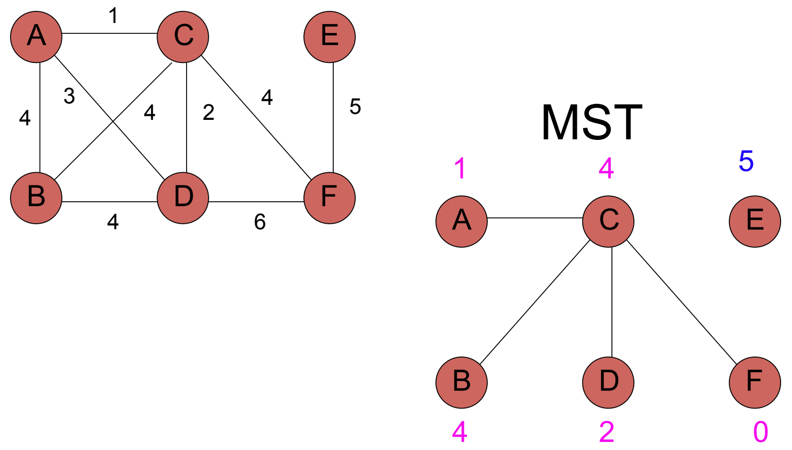

```
while !Empty(H)
    u ← Extract-Min(H)
    visited[u] ← true
    for each edge (u, v) ∈ E
        if !visited[v] and w(u, v) < key(v)
            Decrease-Key(v, w(u, v))
            prev[v] ← u
```



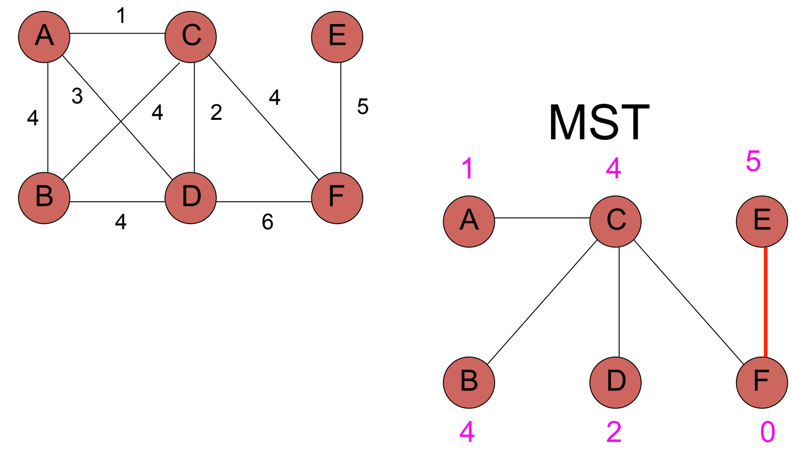

```
while !Empty(H)
    u ← Extract-Min(H)
    visited[u] ← true
    for each edge (u, v) ∈ E
        if !visited[v] and w(u, v) < key(v)
            Decrease-Key(v, w(u, v))
            prev[v] ← u
```



```
while !Empty(H)
        u ← Extract-Min(H)
        visited[u] ← true
        for each edge (u, v) ∈ E
                if !visited[v] and w(u, v) < key(v)
                        Decrease-Key(v, w(u, v))
                        prev[v] ← u
```




```
while !Empty(H)
        u ← Extract-Min(H)
        visited[u] ← true
        for each edge (u, v) ∈ E
                if !visited[v] and w(u, v) < key(v)
                        Decrease-Key(v, w(u, v))
                        prev[v] ← u
```



Correctness of Prim's?

Can we use the min-cut property?

- Given a partion S, let edge *e* be the minimum cost edge that **crosses** the partition. *Every* minimum spanning tree contains edge *e*.

Let S be the set of vertices visited so far

The only time we add a new edge is if it's the lowest weight edge from S to V-S

Running time of Prim' s

```
Prim(G, r)
for all v ∈ V
        key[v] ← ∞
        prev[v] ← null
key[r] ← 0
H ← MakeHeap(key)
while !Empty(H)
        u ← Extract-Min(H)
        visited[u] ← true
        for each edge (u, v) ∈ E
                if !visited[v] and w(u, v) < key(v)
                        Decrease-Key(v, w(u, v))
                        prev[v] ← u
```

Running time of Prim's

```
PRIM(G, r)
for all v ∈ V
        key[v] ← ∞
        prev[v] ← null
key[r] ← 0
H ← MAKEHEAP(key)
while !Empty(H)
        u ← EXTRACT-MIN(H)
        visited[u] ← true
        for each edge (u, v) ∈ E
                if !visited[v] and w(u, v) < key(v)
                        DECREASE-KEY(v, w(u, v))
                        prev[v] ← u
```

Θ(|V|)

Θ(|V|)

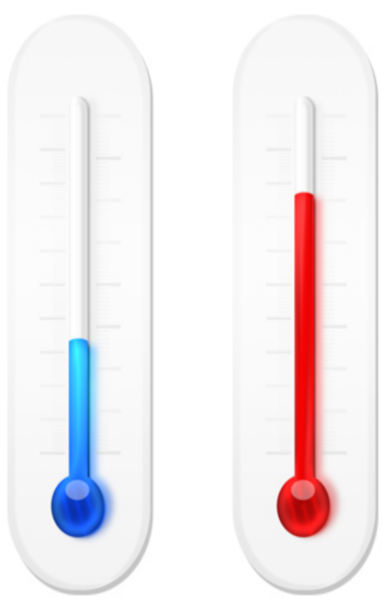
|V| calls to Extract-Min

|E| calls to Decrease-Key

Running time of Prim's

	1 MakeHeap	|V| ExtractMin	|E| DecreaseKey	Total
Array	O(|V|)	$O(|V|^2)$	O(|E|)	$O(|V|^2)$
Bin heap	O(|V|)	O(|V| log |V|)	O(|E| log |V|)	O((|V|+|E|) log |V|) O(|E| log |V|)
Fib heap	O(|V|)	O(|V| log |V|)	O(|E|)	**O(|V| log |V| + |E|)**

Kruskal's: O(|E| log |E|)



Other MST Algorithms

- Boruvka's Algorithm
- Cycle Breaking!
- Many others

Otakar Borůvka

Otakar Borůvka was a Czech mathematician best known today for his work in graph theory, long before this was an established mathematical discipline. Wikipedia

Born: May 10, 1899, Uherský Ostroh

Died: July 22, 1995

You'll meet Boruvka in PS 7b!!

MST Alternative?
Worksheet

Professor Borden proposes a new divide-and-conquer algorithm for computing minimum spanning trees, which goes as follows. Given a graph $G = (V, E)$, partition the set V of vertices into two sets V_1 and V_2 such that $|V_1|$ and $|V_2|$ differ by at most 1. Let E_1 be the set of edges that are incident only on vertices in V_1, and let E_2 be the set of edges that are incident only on vertices in V_2. Recursively solve a minimum-spanning-tree problem on each of the two subgraphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$. Finally, select the minimum-weight edge in E that crosses the cut (V_1, V_2), and use this edge to unite the resulting two minimum spanning trees into a single spanning tree.

Either argue that the algorithm correctly computes a minimum spanning tree of G, or provide an example for which the algorithm fails.