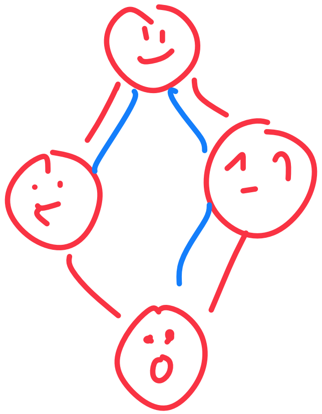
Data Structures

Minimum Spanning Tree

CS 225
Brad Solomon

October 30, 2024



Department of Computer Science

Learning Objectives

Review graph traversal algorithms

Introduce the minimum spanning tree (with weights)

Introduce Kruskal's / Prim's MST Algorithms

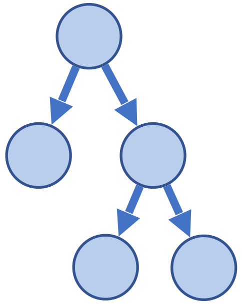
Implement Kruskal's (and potentially Prim's)

Graph Traversals

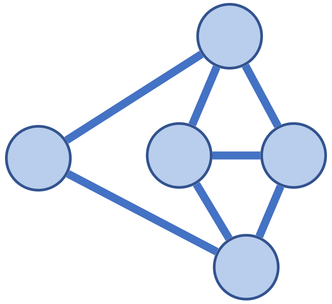
Objective: Visit every vertex and every edge in the graph.

How can we systematically go through a complex graph in the fewest steps?

Tree traversals won't work — lets compare:



- Rooted
- Acyclic
- A clear 'endpoint'



- No root (any start position valid)
- Cycles
- No obvious 'endpoint'

```
BFS(G, v):
  Queue q
  setDist(v, 0)
  q.enqueue(v)

  while !q.empty():
    v = q.dequeue()

    foreach (Vertex w : G.adjacent(v)):
      if( getDist(w) == -1):
          setLabel((v, w), DISCOVERY)
          setPred(w, v)
          setDist(w, v + 1)
          q.enqueue(w)
      else:
          setLabel((v, w), CROSS)
```

Init

New or not

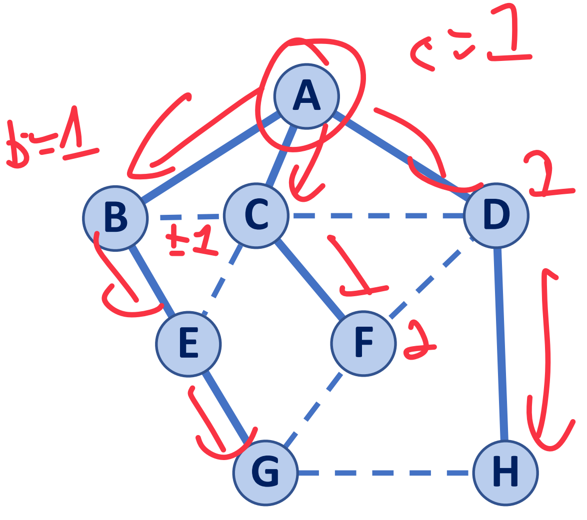
Add to queue

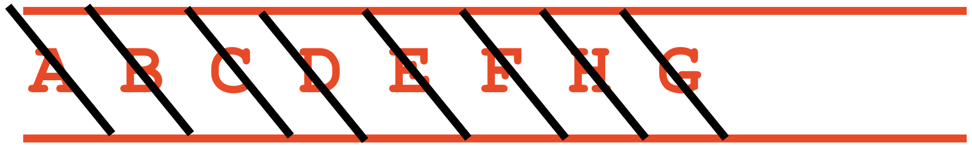
v	d	P	Adjacent Edges
A	0	-	B C D
B	1	A	A C E
C	1	A	A B D E F
D	1	A	A C F H
E	2	B	B C G
F	2	C	C D G
G	3	E	E F H
H	2	D	D G

A - B - C - A

A - B - C - D - A

cross edge





BFS Observations

1. BFS can be used to count components
2. BFS can be used to detect cycles

3. The BFS 'distance' value is always the shortest distance from source to any vertex (and the discovery edges form a MST)

↳ unweighted graphs

4. The endpoints of a cross edge never differ in distance by more than 1 ($|\mathbf{d(u) - d(v)}| = \mathbf{1}$)

```
DFS(G):
  foreach (Vertex v : G.vertices()):
   setPred(v, NULL)
   setDist(v, -1)

  foreach (Edge e : G.edges()):
    setLabel(e, UNEXPLORED)

  foreach (Vertex v : G.vertices()):
    if getDist(v) == -1:
      DFS(G, v)
```

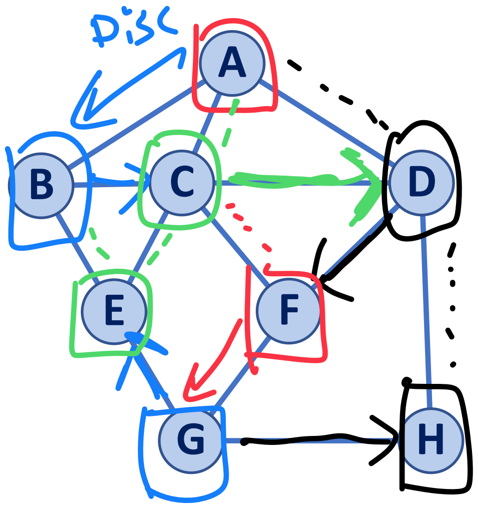
Init

connected components

```
DFS(G, v):

    foreach (Vertex w : G.adjacent(v)):
      if( getDist(w) == -1):
        setLabel((v, w), DISCOVERY)
        setPred(w, v)
        setDist(w, v + 1)
        DFS(G, w)
      else:
        setLabel((v, w), BACK)
```

← No queue, instead stack

← call stack



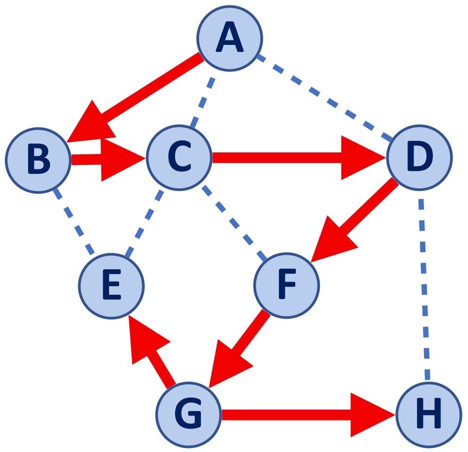
A B C D F G E H

↳ Dont relabel

'if not labeled, set label'

```
DFS(G, v):

    foreach (Vertex w : G.adjacent(v)):
        if( getDist(w) == -1):
            setLabel((v, w), DISCOVERY)
            setPred(w, v)
            setDist(w, v + 1)
            DFS(G, w)
        else:
            setLabel((v, w), BACK)
```

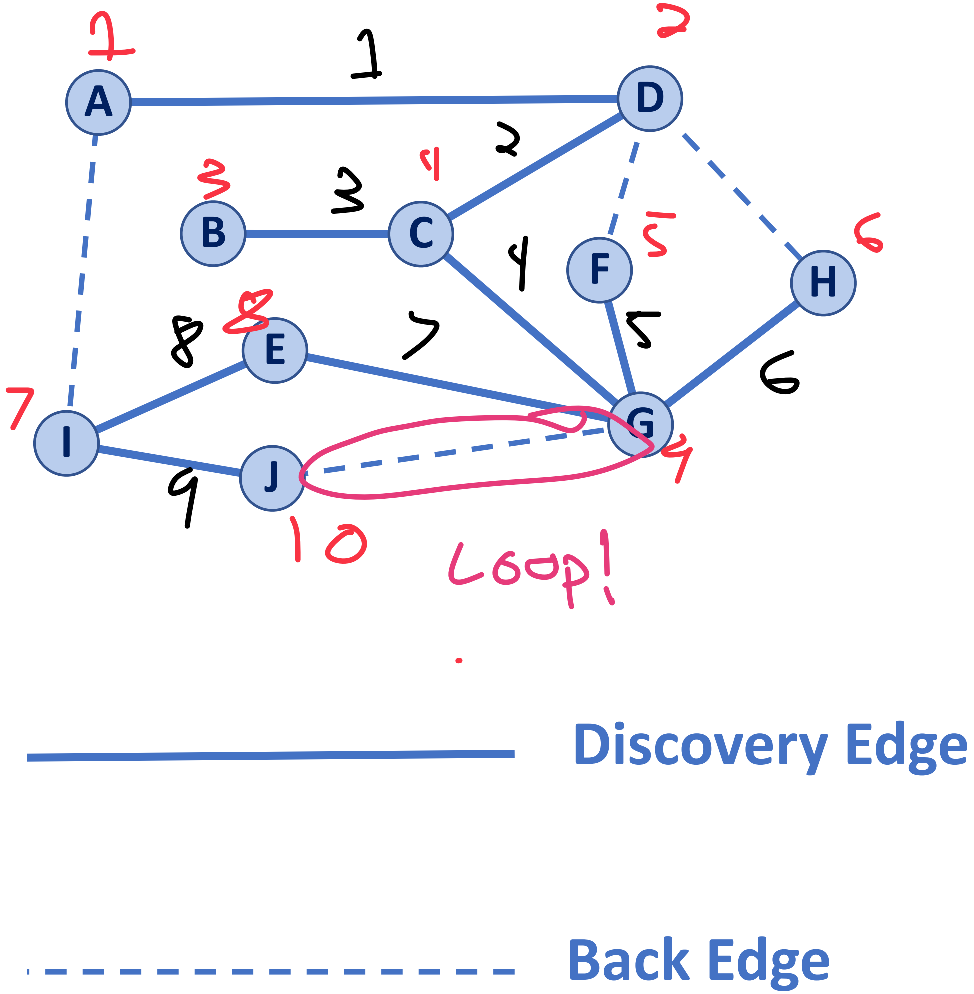
v	d	P	Adjacent Edges
A	0	-	B C D
B	1	A	A C E
C	2	B	A B D E F
D	3	C	A C F H
E	6	G	B C G
F	4	D	C D G
G	5	F	E F H
H	6	G	D G



A B C D F G E H

A → C → E → G

Traversal: DFS



Do we still make a spanning tree?

↳ Yes! ↳ n-1 edges connecting n nodes

Does distance have meaning here?

↳ Not really!

↳ No shortest path soln!

Do our edge labels have meaning here?

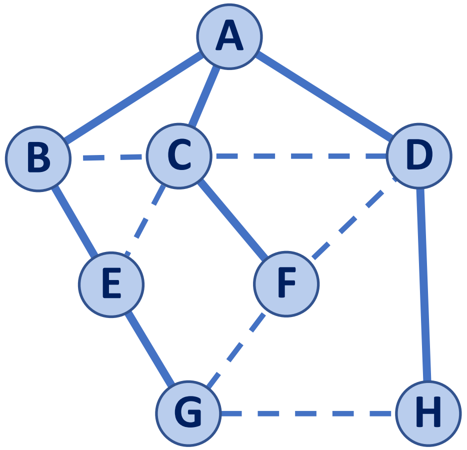
↳ No clear property relating 2 vertices

↳ But still shows cycles

Efficiency: DFS vs BFS

(Traversal) $|V| = n, |E| = m$

BFS: $O(n+m)$



V vertices

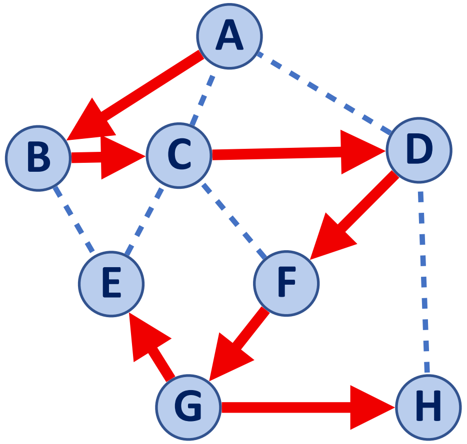
A B C D E F H G

All neighbors

each deg(v)

$\sum_{v} deg(v) = 2|E|$

DFS: $O(n+m)$

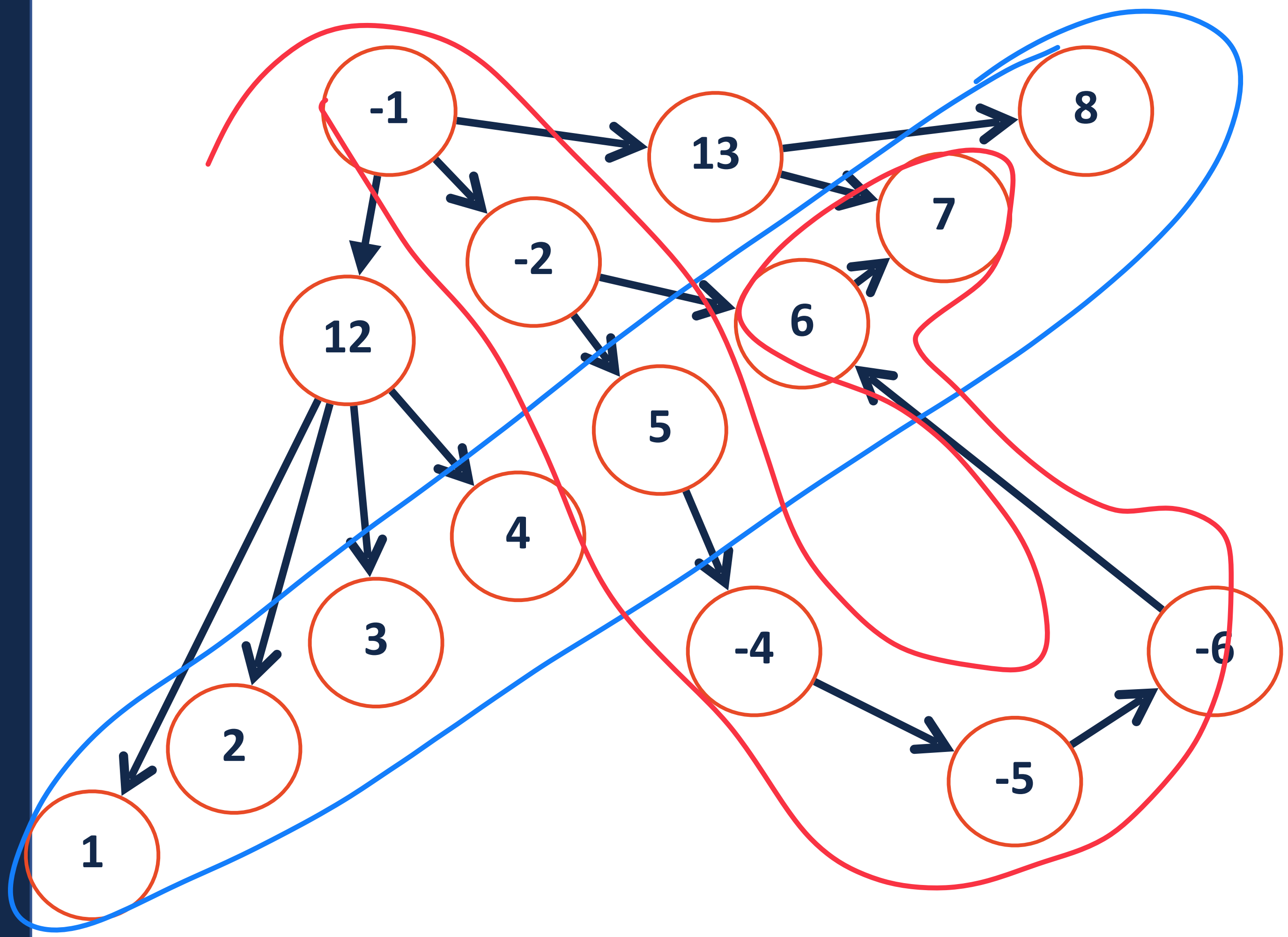


V vertices

A B C D F G E H

each deg(v)

Space Efficiency: DFS vs BFS



BFS stores in queue max level

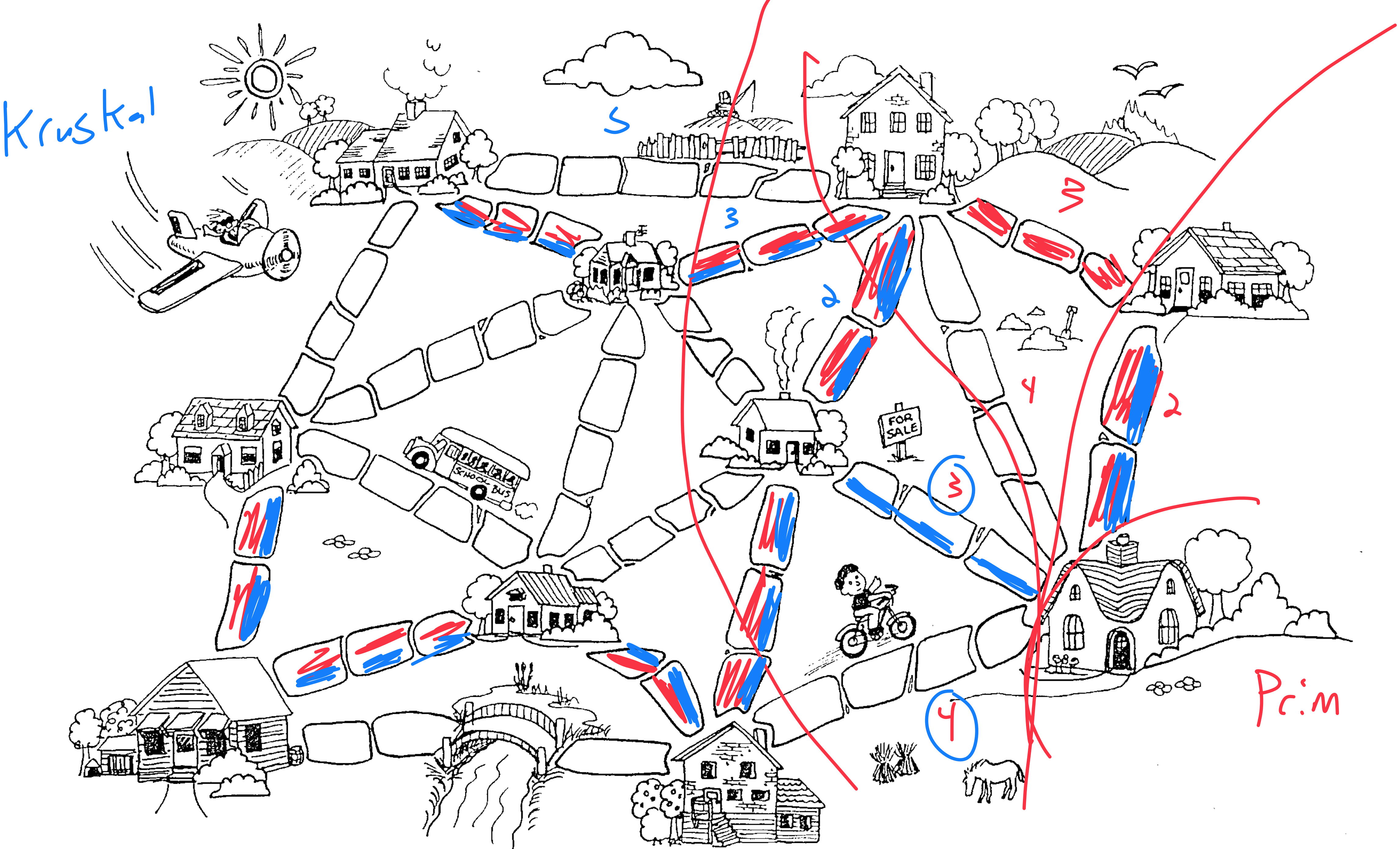
DFS can store longest path

Summary: DFS and BFS

$|V| = n, |E| = m$

Both are **O(n+m)** traversals! They label every edge and every node

BFS	**DFS**
Solves unweighted MST	Solves unweighted MST
Solves shortest path	
Solves cycle detection	Solves cycle detection
Memory bounded by width	Memory bounded by longest path
	↳ considered better in memory


Kruskal
Prim
5
3
3
2
4
2
3
4
FOR SALE
SCHOOL BUS

Minimum Spanning Tree Algorithms

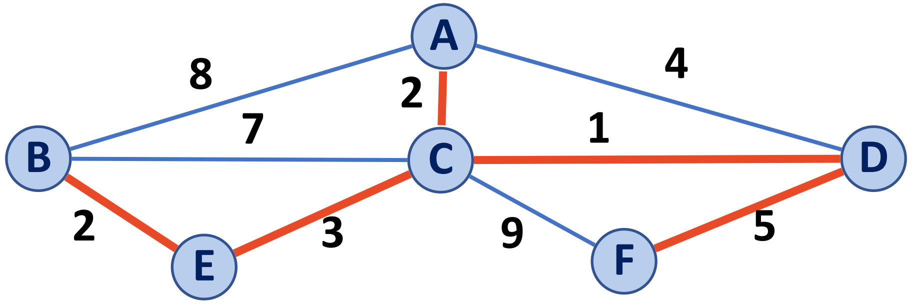
Input: Connected, undirected graph **G** with edge weights (unconstrained, but must be additive)

(4)

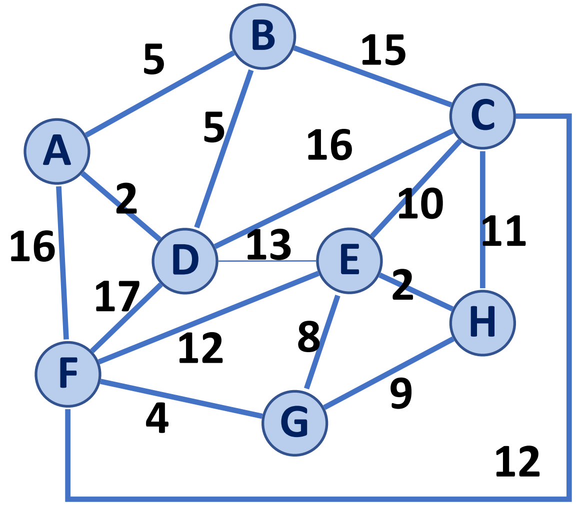
Output: A graph G’ with the following properties:

- G’ is a spanning graph of G
- G’ is a tree (connected, acyclic)
- G’ has a minimal total weight among all spanning trees

G' has all vertices but n-1 edges



Kruskal's Algorithm → Graph soln to MST problem



Adjacency Matrix / List
↳ O(1)

What information do I need?

1) A fast way to get edge weights

1.5) Optimize finding repeated min

↳ knowledge of what edges are

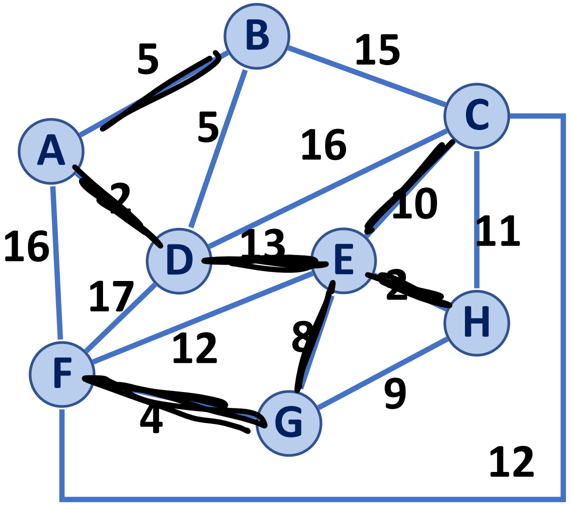
2) A fast way to know if two vertices are connected

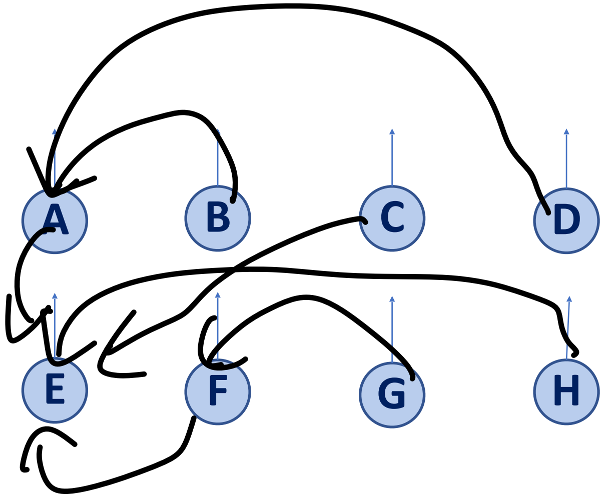
Min heap to solve (1/1.5)

Disjoint set for (2)

Kruskal's Algorithm

(A, D)	✓
(E, H)	✓
(F, G)	✓
(A, B)	✓
(B, D)	✗
(G, E)	✓
(G, H)	✗
(E, C)	✓
(C, H)	✗
(E, F)	✗
(F, C)	✗
(D, E)	✓
(B, C)	
(C, D)	
(A, F)	
(D, F)	





1) Build a **priority queue** on edges

↳ min heap

↳ sorted list

2) Build a **disjoint set** on vertices

↳ All vertices start as own set

3) Repeat take min edge

↳ If connect two sets

↳ union sets

↳ record edge

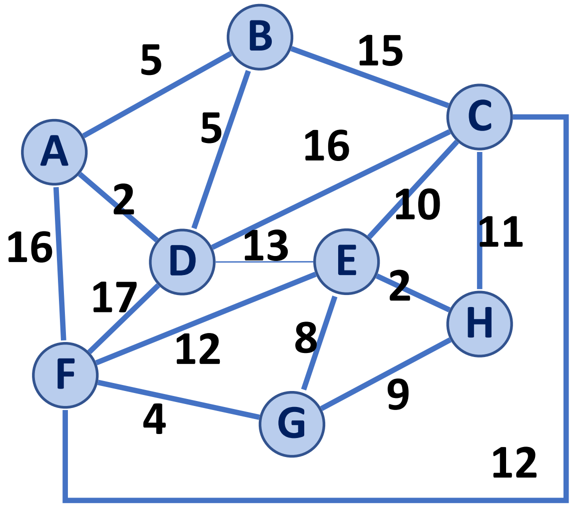
4) Stop when:

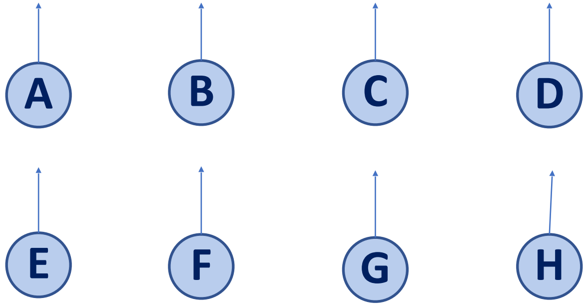
– n-1 nodes recorded

– I have one disjoint set

Kruskal's Algorithm

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






```
KruskalMST(G):
  DisjointSets forest
  foreach (Vertex v : G.vertices()):
    forest.makeSet(v)

  PriorityQueue Q    // min edge weight
  Q.buildFromGraph(G.edges())

  Graph T = (V, {})

  while |T.edges()| < n-1:
    Vertex (u, v) = Q.removeMin()
    if forest.find(u) != forest.find(v):
       T.addEdge(u, v)
       forest.union( forest.find(u),
                     forest.find(v) )

  return T
```

init sets

???

stop case

if 2 sets merge them

Kruskal's Algorithm

Big O?

```
KruskalMST(G):
  DisjointSets forest
  foreach (Vertex v : G.vertices()):
    forest.makeSet(v)

  PriorityQueue Q    // min edge weight
  Q.buildFromGraph(G.edges())

  Graph T = (V, {})

  while |T.edges()| < n-1:
    Vertex (u, v) = Q.removeMin()
    if forest.find(u) != forest.find(v):
       T.addEdge(u, v)
       forest.union( forest.find(u),
                     forest.find(v) )

  return T
```

Kruskal’s Algorithm

Priority Queue:	Heap	Sorted Array
Building :7		
Each removeMin :12		

```
1  KruskalMST(G):
2    DisjointSets forest
3    foreach (Vertex v : G.vertices()):
4      forest.makeSet(v)
5
6    PriorityQueue Q    // min edge weight
7    Q.buildFromGraph(G.edges())
8
9    Graph T = (V, {})
10
11   while |T.edges()| < n-1:
12     Vertex (u, v) = Q.removeMin()
13     if forest.find(u) != forest.find(v):
14        T.addEdge(u, v)
15        forest.union( forest.find(u),
16                      forest.find(v) )
17
18   return T
19
```

Kruskal's Algorithm

Priority Queue:	Total Running Time
Heap	
Sorted Array	

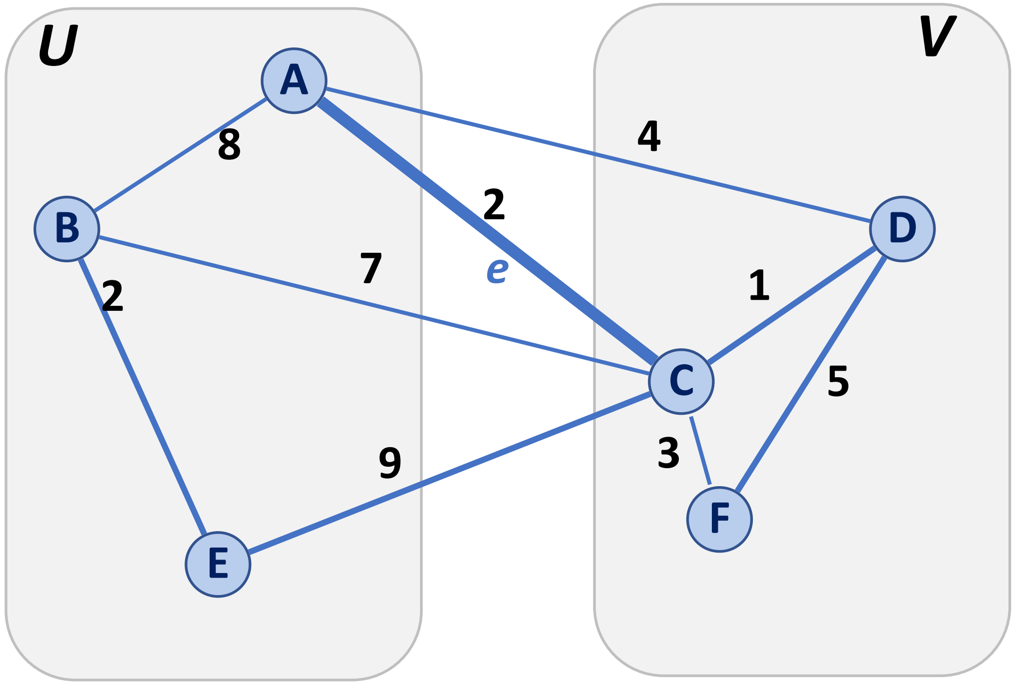
```
KruskalMST(G):
  DisjointSets forest
  foreach (Vertex v : G.vertices()):
    forest.makeSet(v)

  PriorityQueue Q    // min edge weight
  Q.buildFromGraph(G.edges())

  Graph T = (V, {})

  while |T.edges()| < n-1:
    Vertex (u, v) = Q.removeMin()
    if forest.find(u) != forest.find(v):
       T.addEdge(u, v)
       forest.union( forest.find(u),
                     forest.find(v) )

  return T
```

Partition Property

Consider an arbitrary partition of the vertices on **G** into two subsets **U** and **V**.

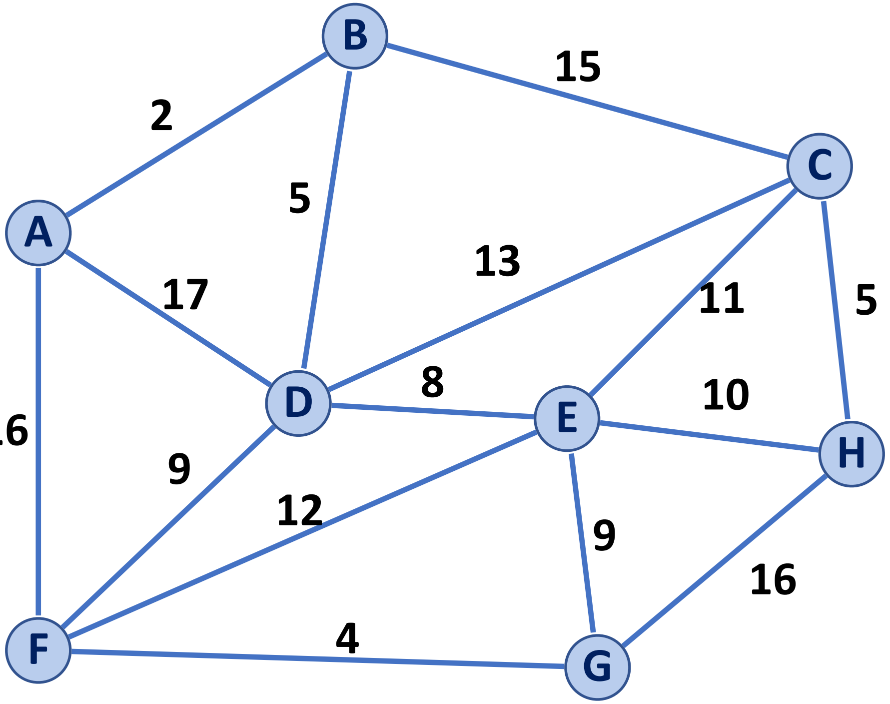
Let **e** be an edge of minimum weight across the partition.

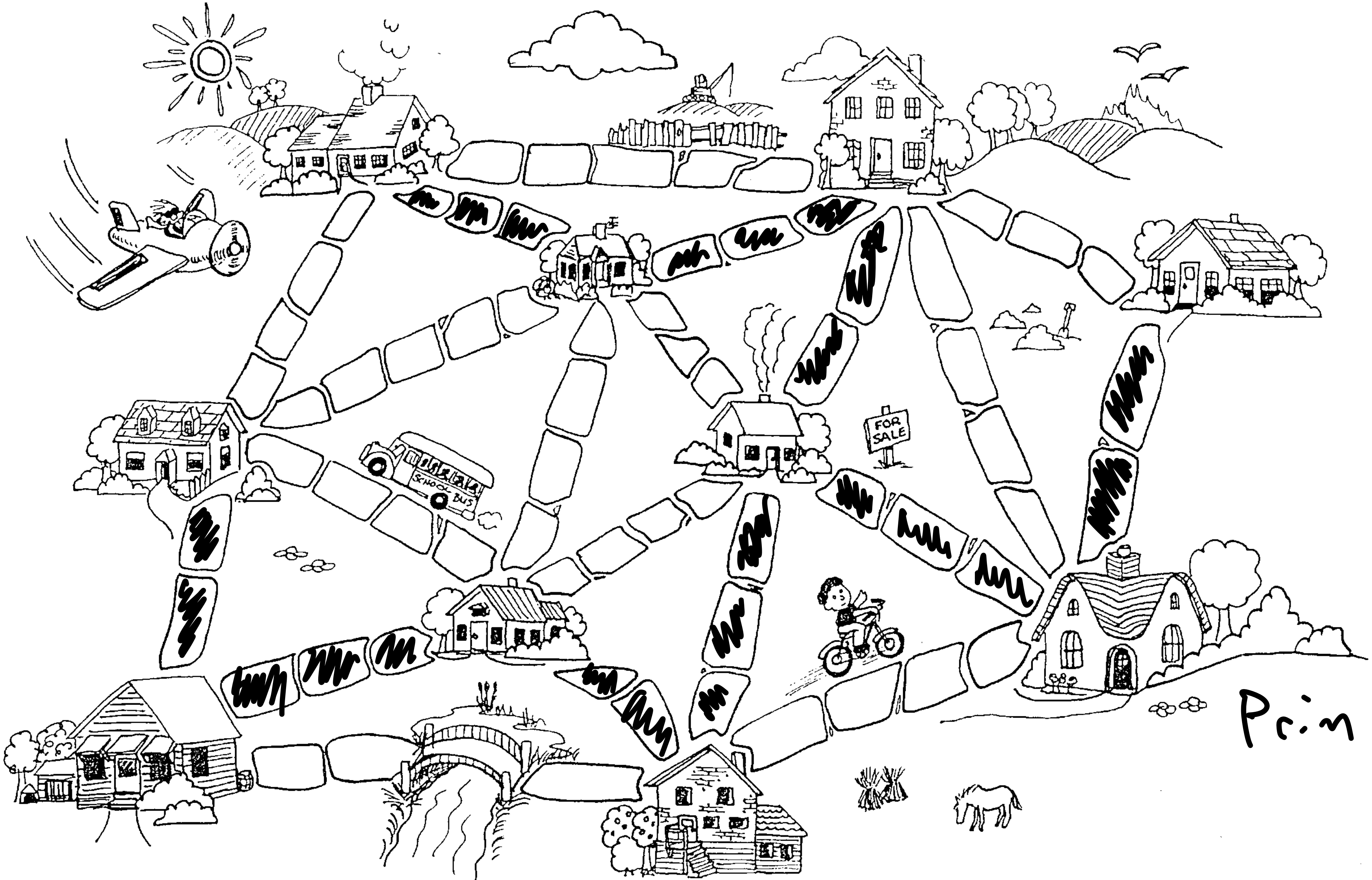
Then **e** is part of some minimum spanning tree.



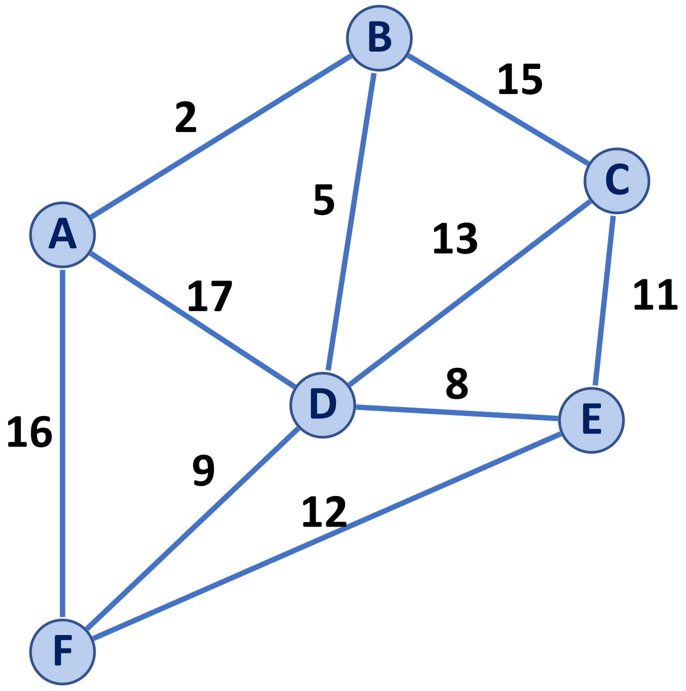
Partition Property

The partition property suggests an algorithm:




FOR SALE
SCHOOL BUS
Prim

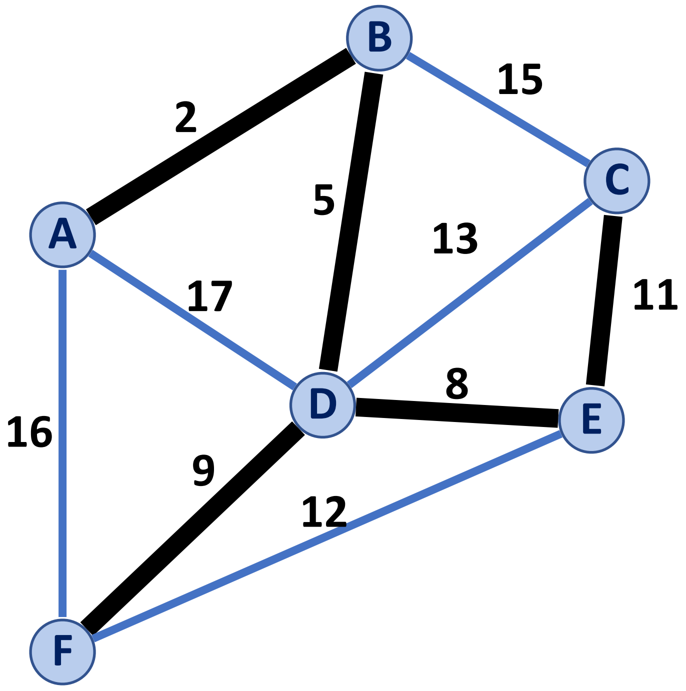
Prim's Algorithm



A	B	C	D	E	F

```
PrimMST(G, s):
  Input: G, Graph;
         s, vertex in G, starting vertex
  Output: T, a minimum spanning tree (MST) of G

  foreach (Vertex v : G.vertices()):
    d[v] = +inf
    p[v] = NULL
  d[s] = 0

  PriorityQueue Q   // min distance, defined by d[v]
  Q.buildHeap(G.vertices())
  Graph T           // "labeled set"

  repeat n times:
    Vertex m = Q.removeMin()
    T.add(m)
    foreach (Vertex v : neighbors of m not in T):
      if cost(v, m) < d[v]:
        d[v] = cost(v, m)
        p[v] = m

  return T
```

Prim's Algorithm



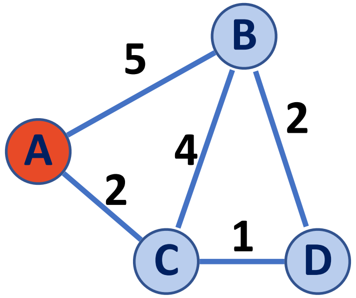
A	B	C	D	E	F
0, —	2, A	11, E	5, B	8, D	9, D

```
PrimMST(G, s):
  Input: G, Graph;
         s, vertex in G, starting vertex
  Output: T, a minimum spanning tree (MST) of G

  foreach (Vertex v : G.vertices()):
    d[v] = +inf
    p[v] = NULL
  d[s] = 0

  PriorityQueue Q   // min distance, defined by d[v]
  Q.buildHeap(G.vertices())
  Graph T           // "labeled set"

  repeat n times:
    Vertex m = Q.removeMin()
    T.add(m)
    foreach (Vertex v : neighbors of m not in T):
      if cost(v, m) < d[v]:
        d[v] = cost(v, m)
        p[v] = m

  return T
```

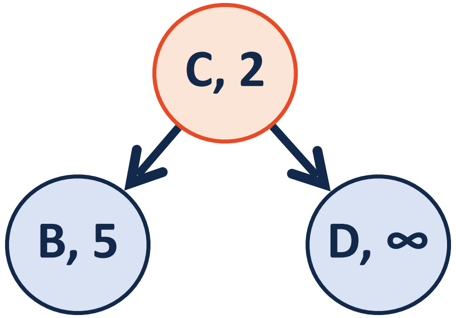
Prim's Big O

```
PrimMST(G, s):
  foreach (Vertex v : G.vertices()):
    d[v] = +inf
    p[v] = NULL
  d[s] = 0

  PriorityQueue Q // min distance, defined by d[v]
  Q.buildHeap(G.vertices())
  Graph T         // "labeled set"

  repeat n times:
    Vertex m = Q.removeMin()
    T.add(m)
    foreach (Vertex v : neighbors of m not in T):
      if cost(v, m) < d[v]:
        d[v] = cost(v, m)
        p[v] = m

```

A	B	C	D
0	5	2	∞

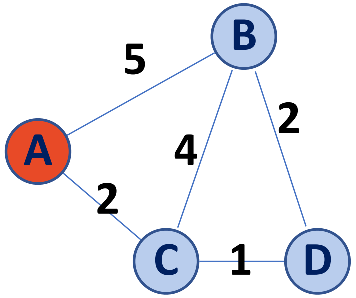



```
PrimMST(G, s):
  foreach (Vertex v : G.vertices()):
    d[v] = +inf
    p[v] = NULL
  d[s] = 0

  PriorityQueue Q // min distance, defined by d[v]
  Q.buildHeap(G.vertices())
  Graph T          // "labeled set"

  repeat n times:
    Vertex m = Q.removeMin()
    T.add(m)
    foreach (Vertex v : neighbors of m not in T):
      if cost(v, m) < d[v]:
        d[v] = cost(v, m)
        p[v] = m

```



	Adj. Matrix	Adj. List
Heap	O(n) + ___________ + O(n^2) + ________________	O(n) + ___________ + O(m) + ________________

(A, 0)
(D, ∞)
(C, 2)
(B, 5)




```
PrimMST(G, s):
  foreach (Vertex v : G.vertices()):
    d[v] = +inf
    p[v] = NULL
  d[s] = 0

  PriorityQueue Q // min distance, defined by d[v]
  Q.buildHeap(G.vertices())
  Graph T          // "labeled set"

  repeat n times:
    Vertex m = Q.removeMin()
    T.add(m)
    foreach (Vertex v : neighbors of m not in T):
      if cost(v, m) < d[v]:
        d[v] = cost(v, m)
        p[v] = m
```

	Adj. Matrix	Adj. List
Heap	$O(n^2 + m \lg(n))$	$O(n \lg(n) + m \lg(n))$
Unsorted Array		

Prim's Algorithm

Sparse Graph:

Dense Graph:

```
PrimMST(G, s):
  foreach (Vertex v : G.vertices()):
    d[v] = +inf
    p[v] = NULL
  d[s] = 0

  PriorityQueue Q // min distance, defined by d[v]
  Q.buildHeap(G.vertices())
  Graph T          // "labeled set"

  repeat n times:
    Vertex m = Q.removeMin()
    T.add(m)
    foreach (Vertex v : neighbors of m not in T):
      if cost(v, m) < d[v]:
        d[v] = cost(v, m)
        p[v] = m
```

	Adj. Matrix	Adj. List
Heap	$O(n^2 + m \lg(n))$	$O(n \lg(n) + m \lg(n))$
Unsorted Array	$O(n^2)$	$O(n^2)$

MST Algorithm Runtime:

Kruskal's Algorithm:

O(n + m log (n))

Prim's Algorithm:

O(n log(n) + m log (n))

Sparse Graph:

Dense Graph:

Suppose I have a new heap:

	Binary Heap	Fibonacci Heap
Remove Min	O(lg(n))	O(lg(n))
Decrease Key	O(lg(n))	O(1)*

What's the updated running time?

```
PrimMST(G, s):
  foreach (Vertex v : G.vertices()):
    d[v] = +inf
    p[v] = NULL
  d[s] = 0

  PriorityQueue Q // min distance, defined by d[v]
  Q.buildHeap(G.vertices())
  Graph T          // "labeled set"

  repeat n times:
    Vertex m = Q.removeMin()
    T.add(m)
    foreach (Vertex v : neighbors of m not in T):
      if cost(v, m) < d[v]:
        d[v] = cost(v, m)
        p[v] = m
```