Data Structures

Traversals 2 and Minimum Spanning Tree

CS 225
Brad Solomon

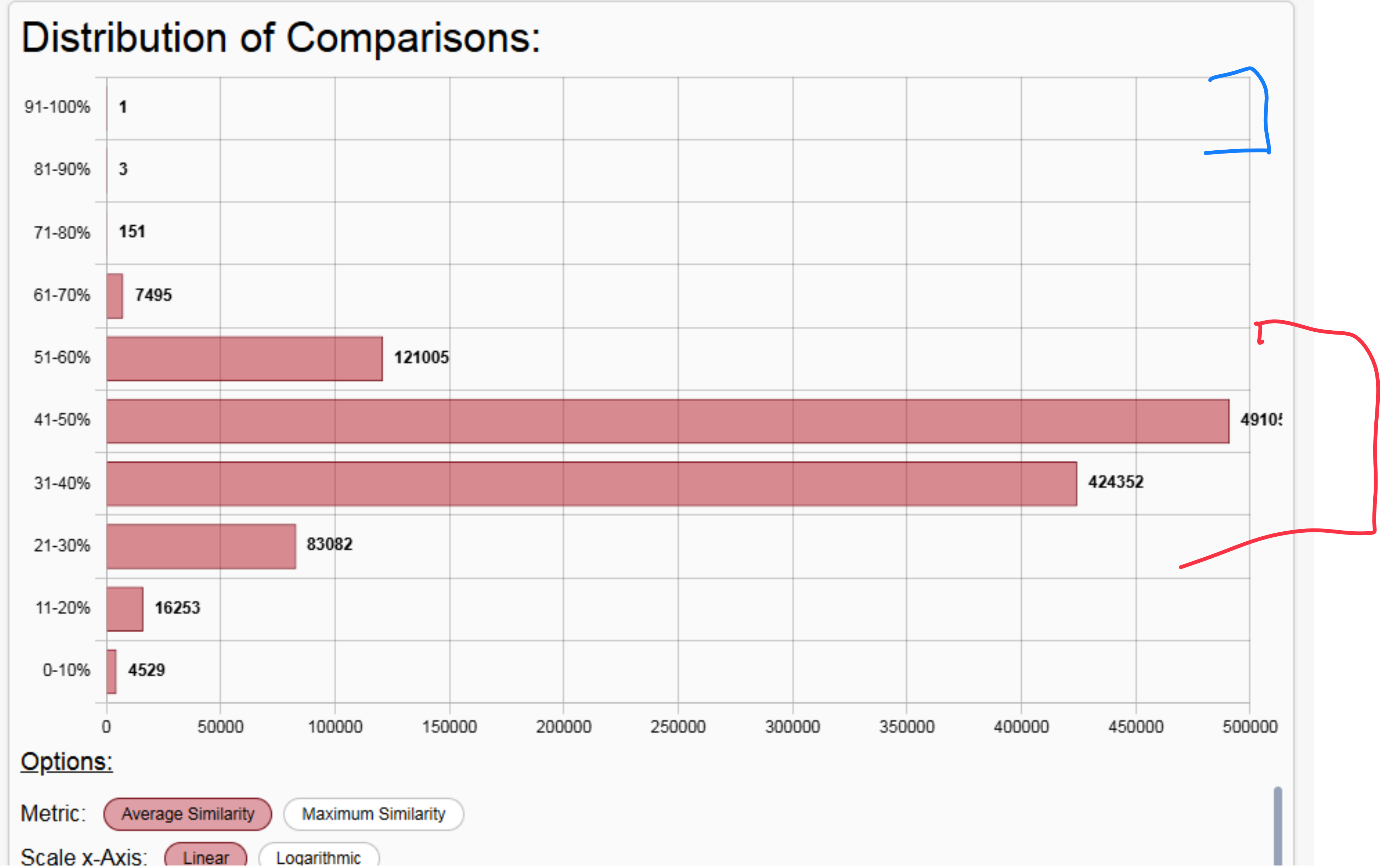
October 31, 2025

Happy Halloween!

^_^

UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN
Department of Computer Science

Great work — keep it up!



Learning Objectives

Review graph traversal algorithms (mostly DFS)

Introduce the minimum spanning tree (with weights)

Introduce Kruskal's / Prim's MST Algorithms

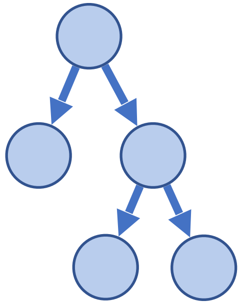
Begin discussing implementation of Kruskal's

Graph Traversals

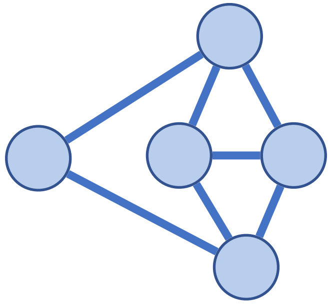
Objective: Visit every vertex and every edge in the graph.

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)
```

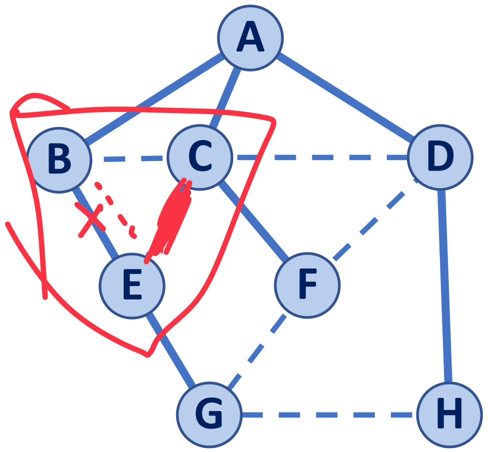
Initialize

label each item

tells us vertex is unlabeled

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

Initialize



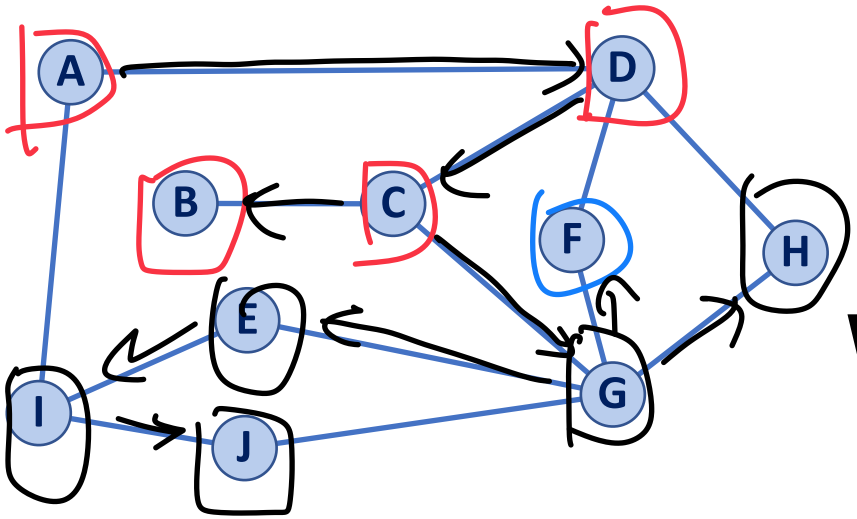
BFS Observations

1. BFS can be used to count components
2. BFS can be used to detect cycles* Not an accurate count

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

4. The endpoints of a cross edge never differ in distance by more than 1 (**|d(u) - d(v)| = 1**) Trivia

Traversal: DFS



Initialize dist / pred / stack

All dist null (start node dist 0)

All pred -1 (start node pred -1)

Stack loaded with start node

While stack not empty

tmp=stack.peek()

Process one child of tmp

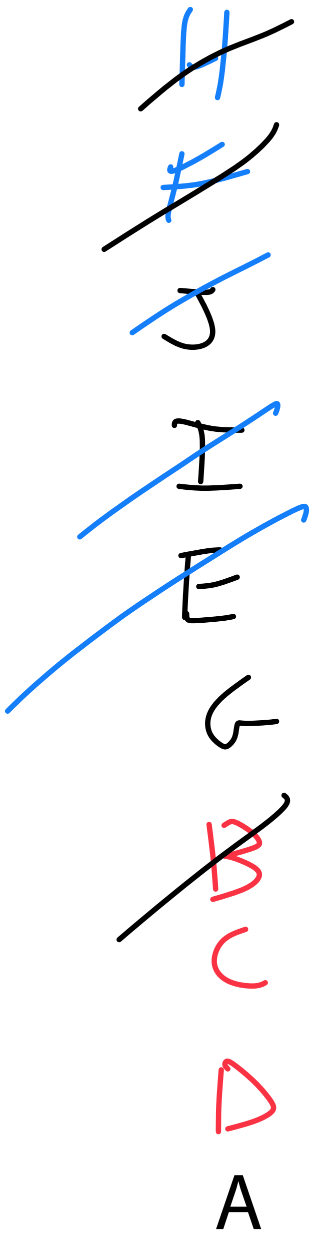
Add to stack

dist = tmp.dist+1

pred = tmp

If no unvisited children

stack.pop()



Stack

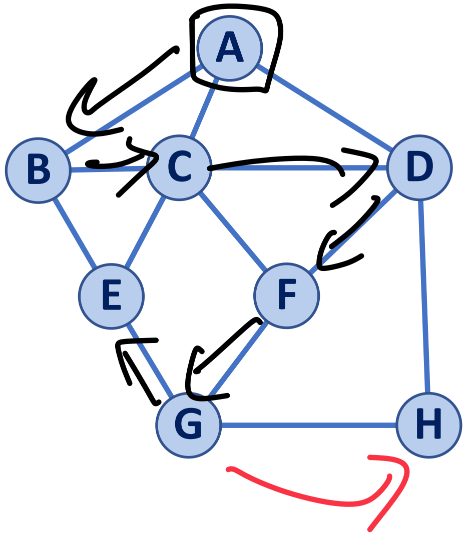
```
DFS(G):
  foreach (Vertex v : G.vertices()):
   setPred(v, NULL)
   setDist(v, -1)

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

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

```
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)
```

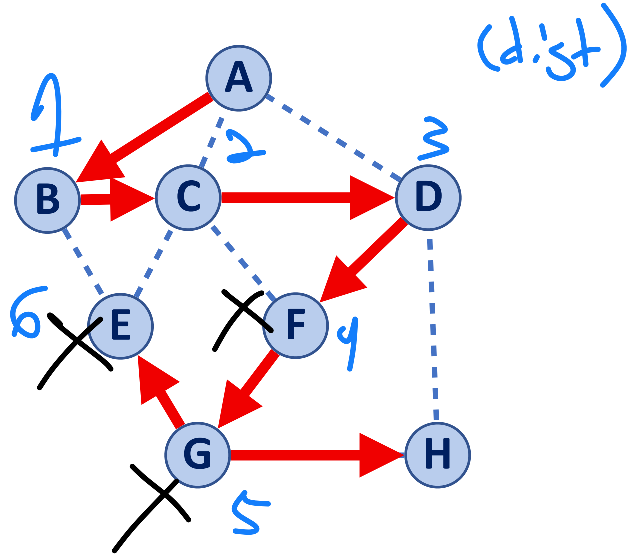


H
~~E~~
G
F
D
C
B
A

```
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	~~4~~ 6	~~D~~ G	D G



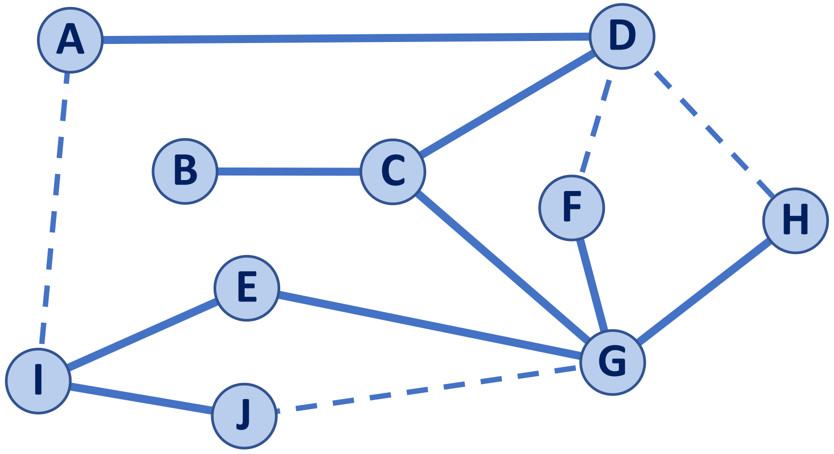
State of stack when H is added:

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

This slide is wrong!

Traversal: DFS

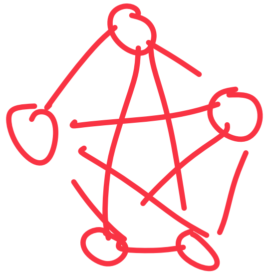
1 discovery edge



Discovery Edge

Back Edge

Do we still make a spanning tree?



↳ connects all vertices

↳ has no cycles

n-1 edges for n vertices

Yes!

Does distance have meaning here?

↳ No clear meaning

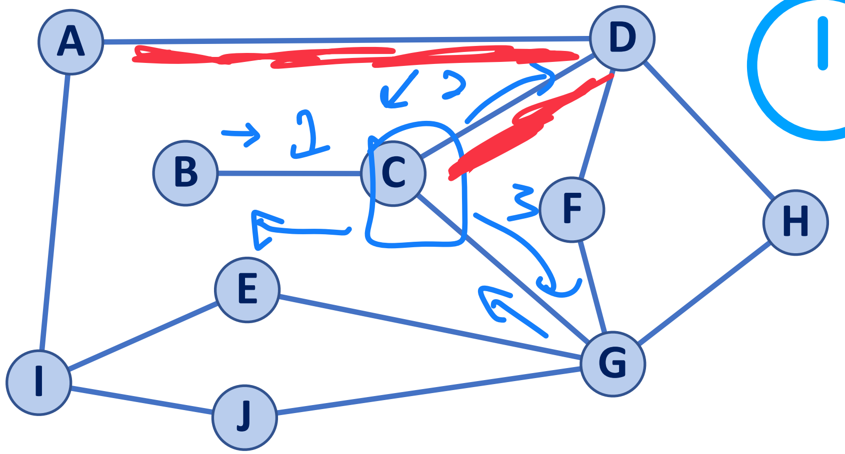
Do our edge labels have meaning here?

↳ Back edges == cross edge

↳ Tells us cycles!

Running time of DFS

$|V| = n \quad |E| = m$



Discussed on webpages

Labeling: $O(n+m)$

- Vertex: w/ dist / pred $\quad O(n)$
  ↳ Do this for every vertex
- Edge: w/ discovery or back $\quad O(m)$
  ↳ Do this for every edge

Important for undirected graphs

Extra work in

Traversal: $O(m)$

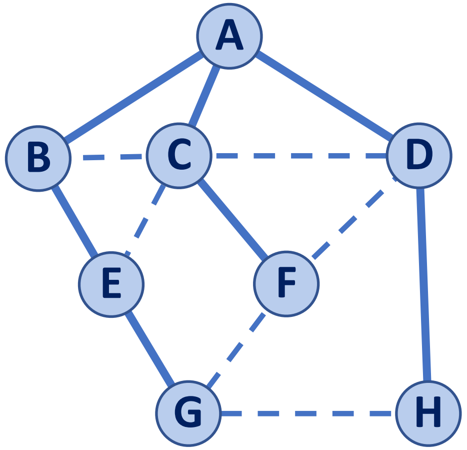
- Vertex: Each vertex looked at / up $\deg(v)$ times $\quad \boxed{\sum_v \deg(v) = 2m}$
- Edge: Every edge visited twice

Total work: $\boxed{O(n+m)}$

Efficiency: DFS vs BFS

BFS: $O(n+m)$

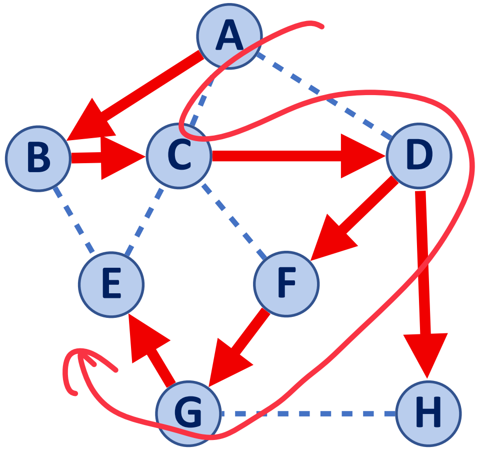
max width



A B C D E F H G

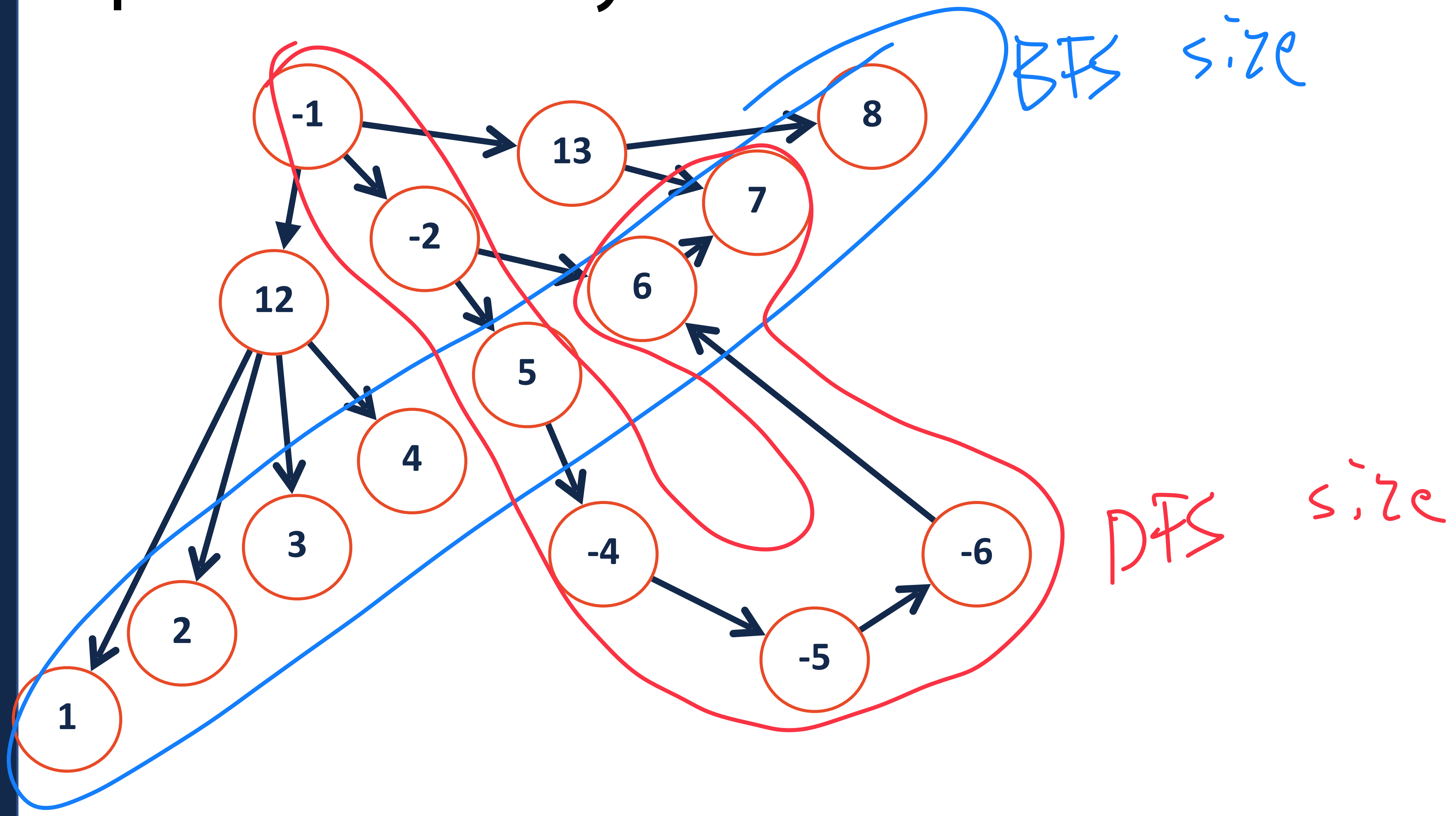
DFS: $O(n+m)$

Longest path



A B C D F G E H

Space Efficiency: DFS vs BFS



Summary: DFS and BFS

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

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

BFS

Solves unweighted MST

Solves shortest path

Solves cycle detection

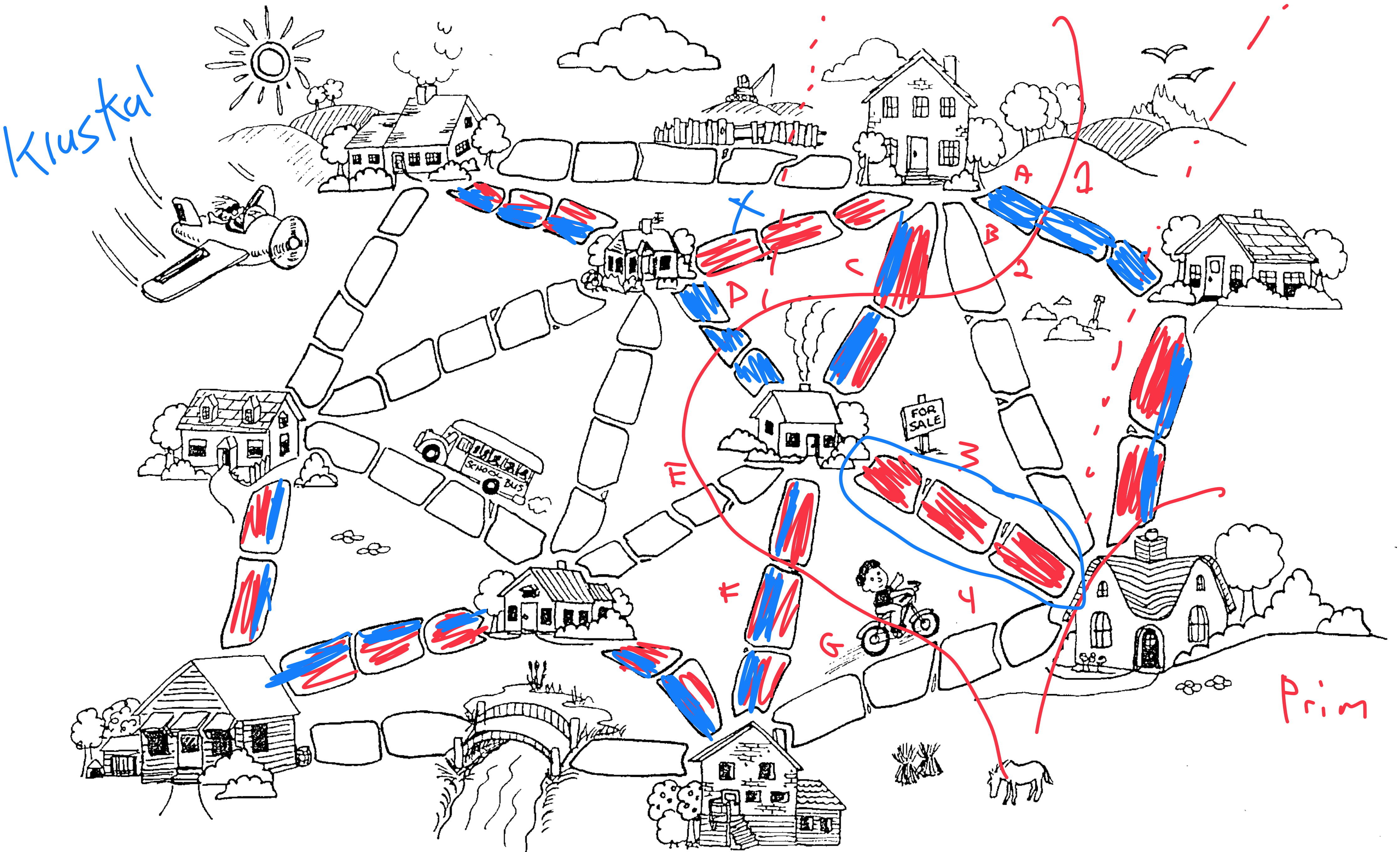
Memory bounded by width

DFS

Solves unweighted MST

Solves cycle detection

Memory bounded by longest path

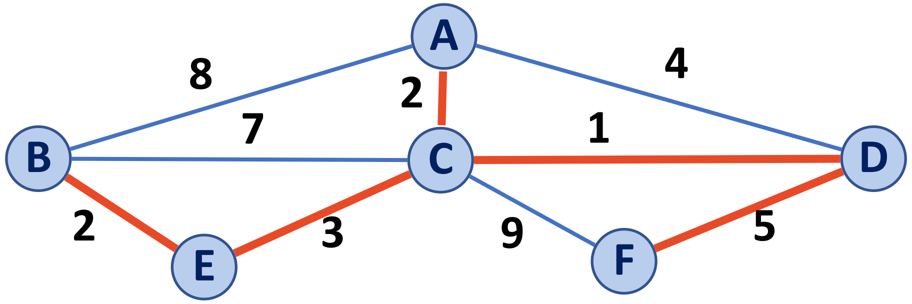

Klustal
FOR SALE
SCHOOL BUS
Prim

Minimum Spanning Tree Algorithms

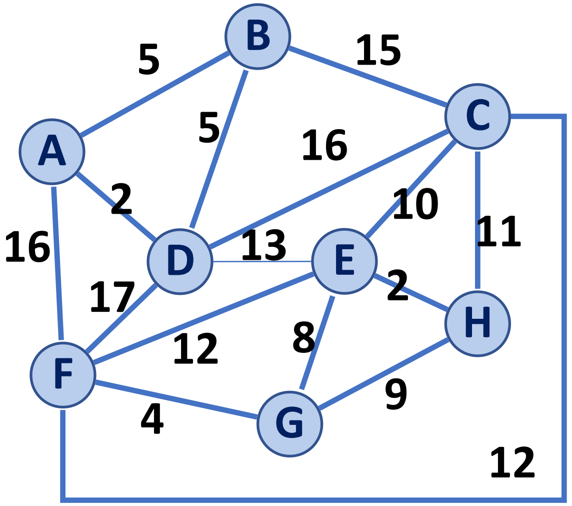
Input: Connected, undirected graph **G** with edge weights (unconstrained, but must be additive)

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



Kruskal's Algorithm *(A graph algorithm for the MST problem)*



What information do I need to get efficiently?

1) The global minimum edge weight

2) Edge connections given a vertex

Not 2

Put

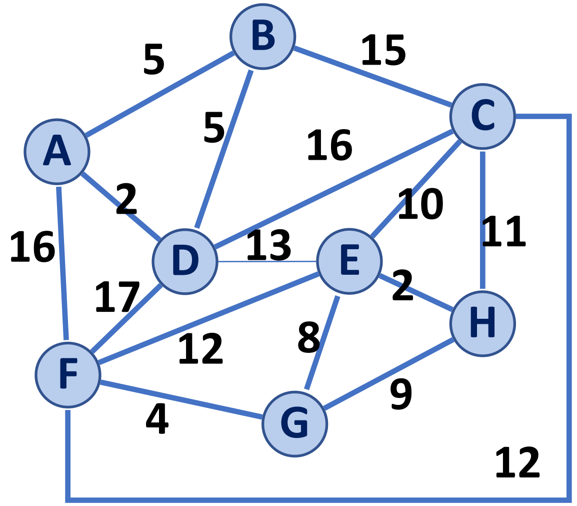
3) If two vertices are already connected

4) A visited list of vertices

↳ Not relevant

connections matter!

Kruskal's Algorithm *(A graph algorithm for the MST problem)*



What information do I need to get efficiently?

1) The global minimum edge weight

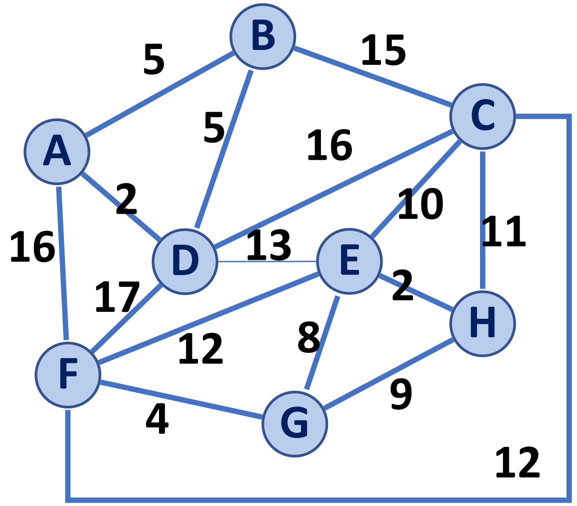
3) If two vertices are already connected

What does this tell me about my implementation choices?

(1) ↳ use a minheap for min edge weight! (priority queue)

(3) ↳ Disjoint set

Kruskal's Algorithm *(A graph algorithm for the MST problem)*



What information do I need to get efficiently?

1) The global minimum edge weight

3) If two vertices are already connected

What does this tell me about my implementation choices?

1) We need a **priority queue** of edges (sorted by weight)

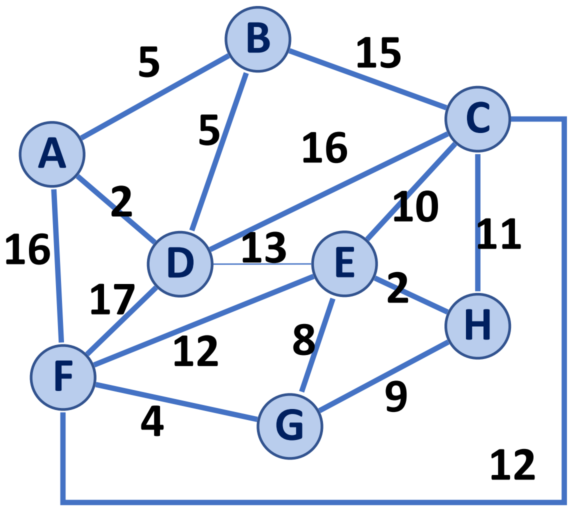
2) We need a **disjoint set** of vertices

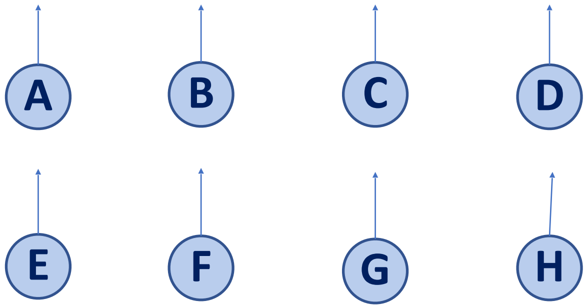
Kruskal’s Algorithm

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

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

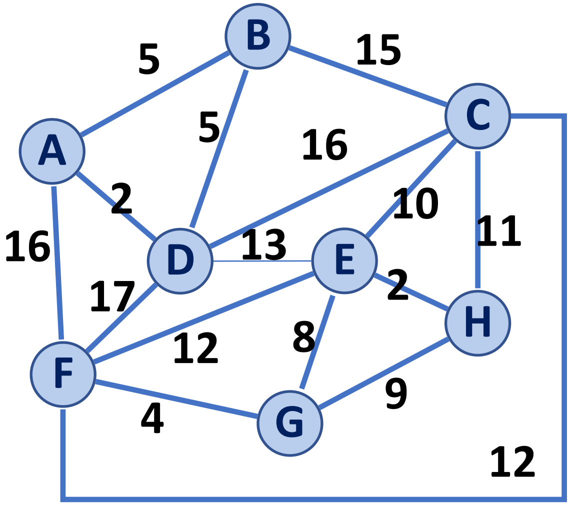
(A, D)
(E, H)
(F, G)
(A, B)
(B, D)
(G, E)
(G, H)
(E, C)
(C, H)
(E, F)
(F, C)
(D, E)
(B, C)
(C, D)
(A, F)
(D, F)

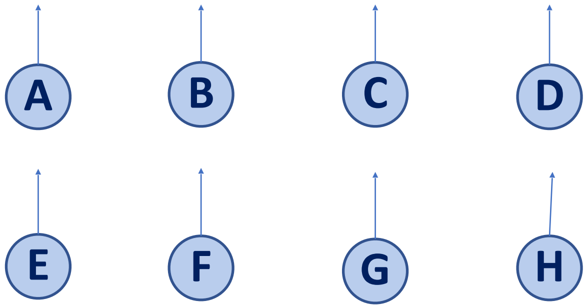




Kruskal's Algorithm

(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

A minheap

or

A sorted array

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

All vertices start as their own set

3) Loop through min edges

If edge connects two disjoint sets

Union sets and record edge in MST

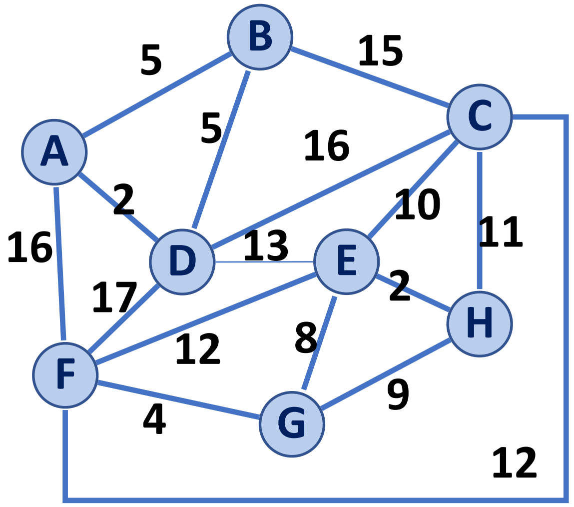
4) Stop when:

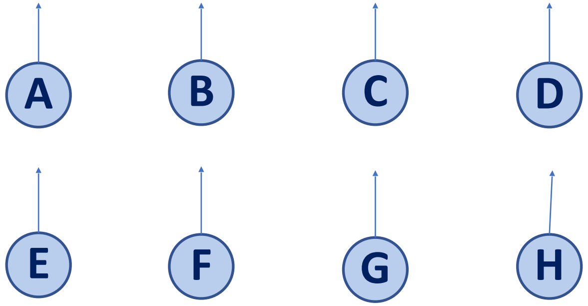
N-1 edges recorded

Only a single disjoint set remains

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
```

Kruskal's Algorithm

```
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
```

|V|= n,|E|= m

What is the Big O?

Kruskal's Algorithm

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

```
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
```

What is the Big O?

2 — 4: O(n)

6 — 7: Heap: O(m)
Sorted List: O(m log m)

11: m x <12-17>

12—17: Heap: O(log m)
Sorted List: O(1)

Disjoint set we treat as O(1) b/c path compression w/ smart union

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:	Heap	Sorted Array
Building :7	O(m)	O(m log m)
Each removeMin :12	O(m log m)	O(m)

Both result in m + m log m

Why is heap good?

If edge weights can change!

```
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
```

Why is sorted array good?

Sorted array not destroyed and can be useful in other algorithms!

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
```

Kruskal's Algorithm

Priority Queue:	Total Running Time
Heap	O(n)+O(m) + O(m log m)
Sorted Array	O(n) + O(m log m) + O(m)

```
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
```

Neat trick: O(log m) = O(log n)

Bounds of n & m relationship?

$$n-1 \leq m \leq n^2$$

Connected graph *Complete graph*