Data Structures

Traversals 2 and Minimum Spanning Tree

CS 225

Brad Solomon

April 6, 2026

UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

Department of Computer Science

Poetry Club

Tired of segfaults and memory errors? Need a break from studying, or interested in the humanities?

- *Tuesday, April 7 6-7PM*
- *English Building 149*
- *Snacks Provided*
- *Fill out QR for snack headcount*

Learning Objectives

Continue 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

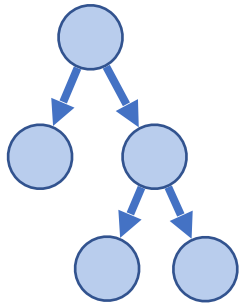
Graph Traversals

we label these

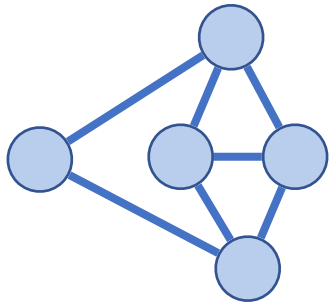
Objective: Visit every vertex and every edge in the graph.

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'

```
12 BFS(G, v):
13   Queue q
14   setDist(v, 0)
15   q.enqueue(v)
16
17   while !q.empty():
18     v = q.dequeue()
19
20     foreach (Vertex w : G.adjacent(v)):
21       if( getDist(w) == -1):
22         setLabel((v, w), DISCOVERY)
23         setPred(w, v)
24         setDist(w, v + 1)
25         q.enqueue(w)
26       else:
27         setLabel((v, w), CROSS)
```

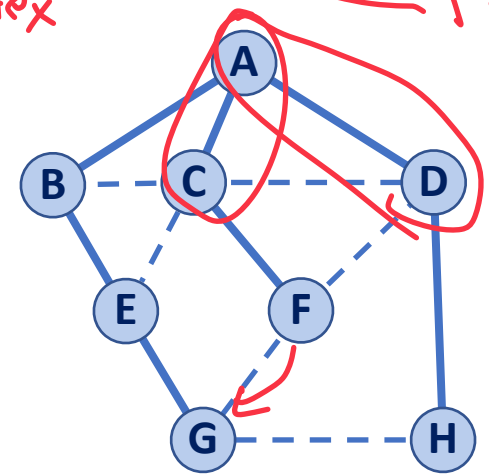
vertex labels

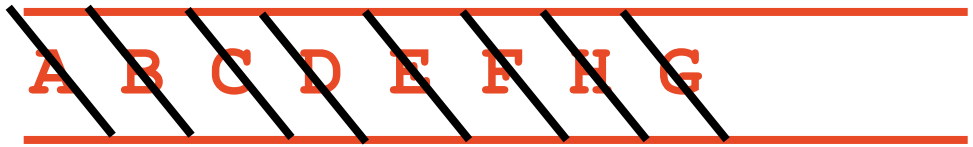
labels

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

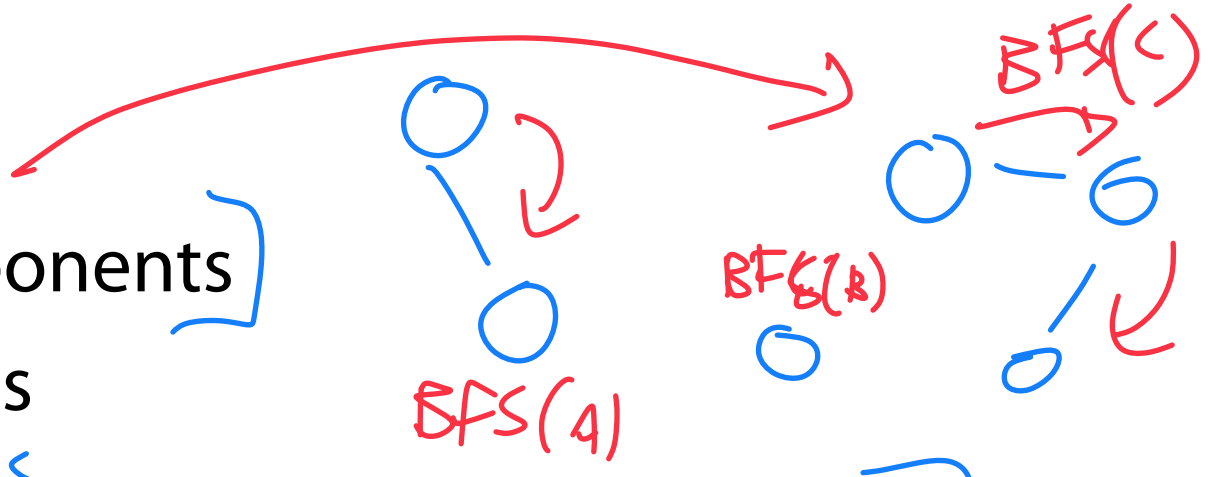
Discovery is new vertex
↳ Have not visited

Cross - Two things (Vertices) we have visited





BFS Observations



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

↳ edge labels

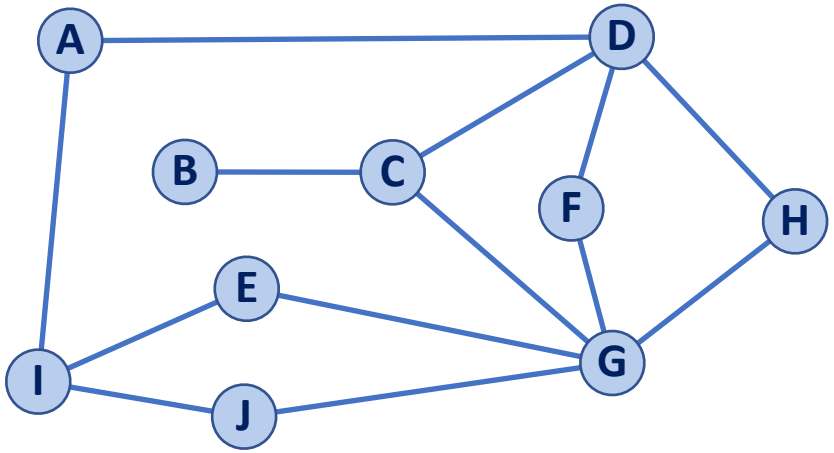
3. The BFS 'distance' value is always the shortest distance from source to any vertex (and the discovery edges form a MST)

↳ vertex labels

edge labels

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

Traversal: DFS



Lets use stack not queue

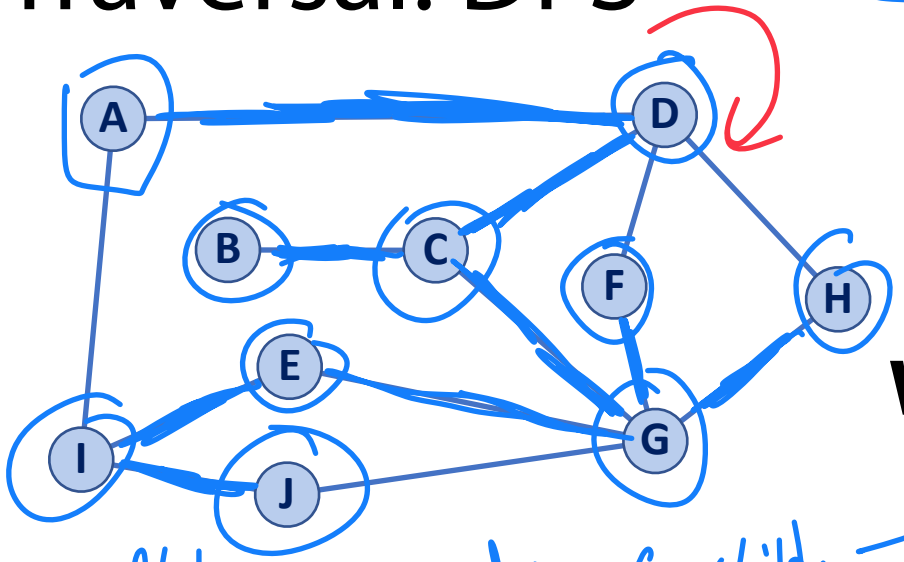
↳ Do the same general concept

1) Initialize start/labels/...

2) While stack is not empty
   ↳ get top of stack
   Process 1 child
   ↳ Add child to stack

3) If top has no unvisited children POP!

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() or top()

Process one child of tmp

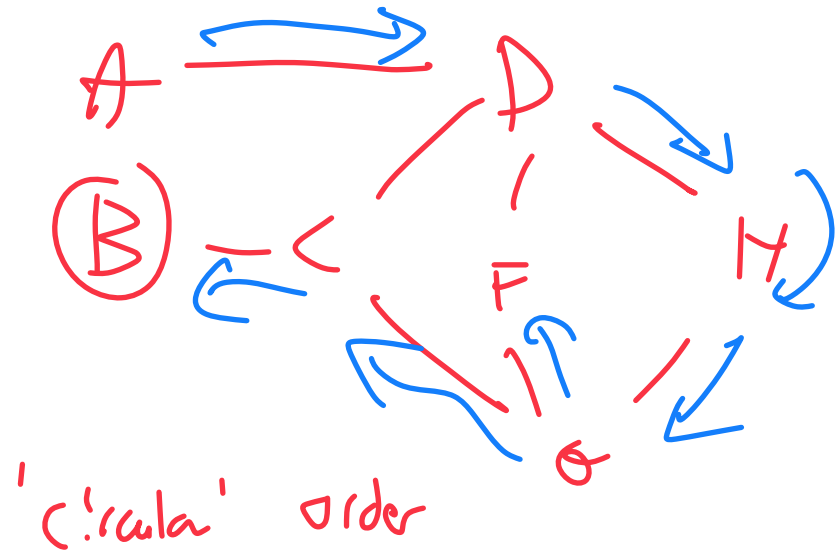
Add to stack

dist = tmp.dist+1

pred = tmp

If no unvisited children

stack.pop()

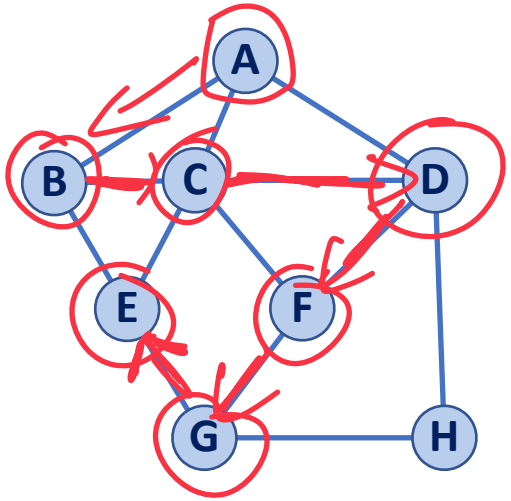


H
F
J
I
E
G
B
C
D
A

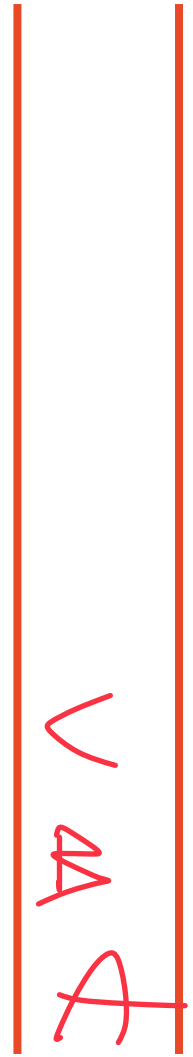
Stack

```
1  DFS(G):
2    foreach (Vertex v : G.vertices()):
3      setPred(v, NULL)
4      setDist(v, -1)
5
6    foreach (Edge e : G.edges()):
7      setLabel(e, UNEXPLORED)
8
9    foreach (Vertex v : G.vertices()):
10     if getDist(v) == -1:
11       DFS(G, v)
```

```
12 DFS(G, v):
13
14     foreach (Vertex w : G.adjacent(v)):
15       if( getDist(w) == -1):
16         setLabel((v, w), DISCOVERY)
17         setPred(w, v)
18         setDist(w, v + 1)
19         DFS(G, w)
20       else:
21         setLabel((v, w), BACK)
```

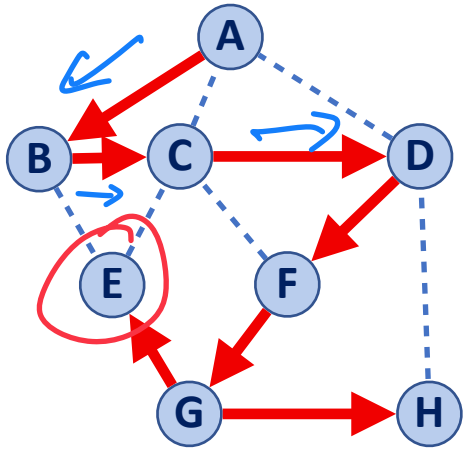


In red are the differences from BFS → DFS



call stack

```
12 DFS(G, v):
13
14     foreach (Vertex w : G.adjacent(v)):
15         if( getDist(w) == -1):
16             setLabel((v, w), DISCOVERY)
17             setPred(w, v)
18             setDist(w, v + 1)
19             DFS(G, w)
20         else:
21             setLabel((v, w), BACK)
```



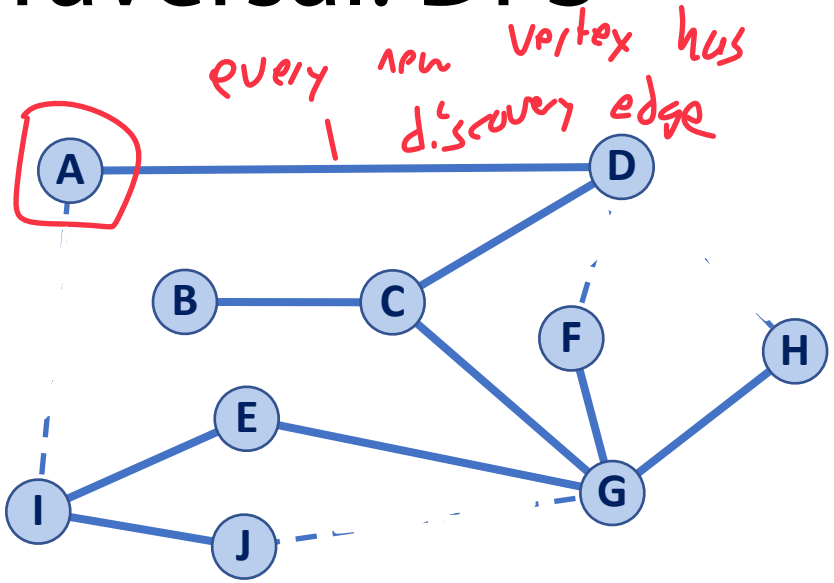
v	d	P	Adjacent Edges
A	0	-	B C D
B	1	A	A C E
C	2	B	A B D E F
D	3	C	A C F H
E	6	G	B C G
F	4	D	C D G
G	5	F	E F H
H	6	G	D G

why this order?
Its arbitrary!

State of stack when H is added:

H
E
G
F
D
C
B
A

Traversal: DFS



Do we still make a spanning tree?

↳ A tree has no cycles
↳ A tree is connected

This means n-1 edges

↳ Yes, discovery edges still form a ST

Does distance have meaning here?

↳ Not in terms of shortest path!

Discovery Edge

Back Edge

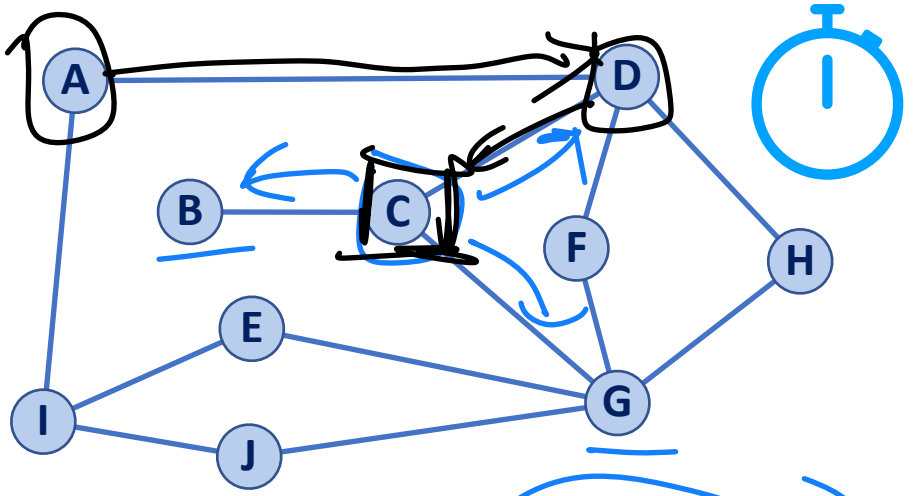
Do our edge labels have meaning here?

↳ Will yes! Back ≡ cross

(DFS) (BFS)

↳ Still tell us cycles

Running time of DFS



Labeling: $O(n+m)$

- Vertex: As dist / prev $O(n)$
- Edge: As discovery (as back $O(m)$

$O(n+m)$

Traversal: $O(m)\in \rightarrow O(n+m)$

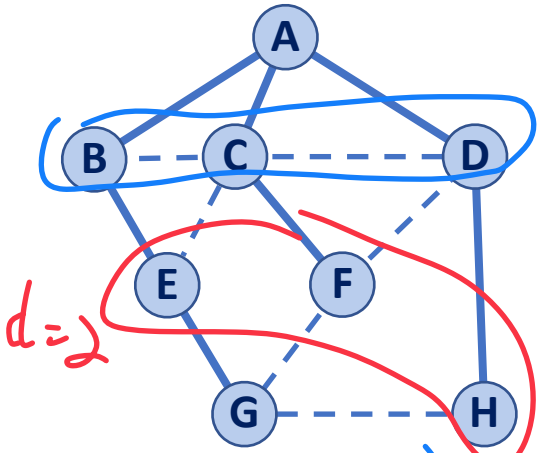
- Vertex: Every vertex is visited once & looks at all children
- Edge: Each are viewed twice

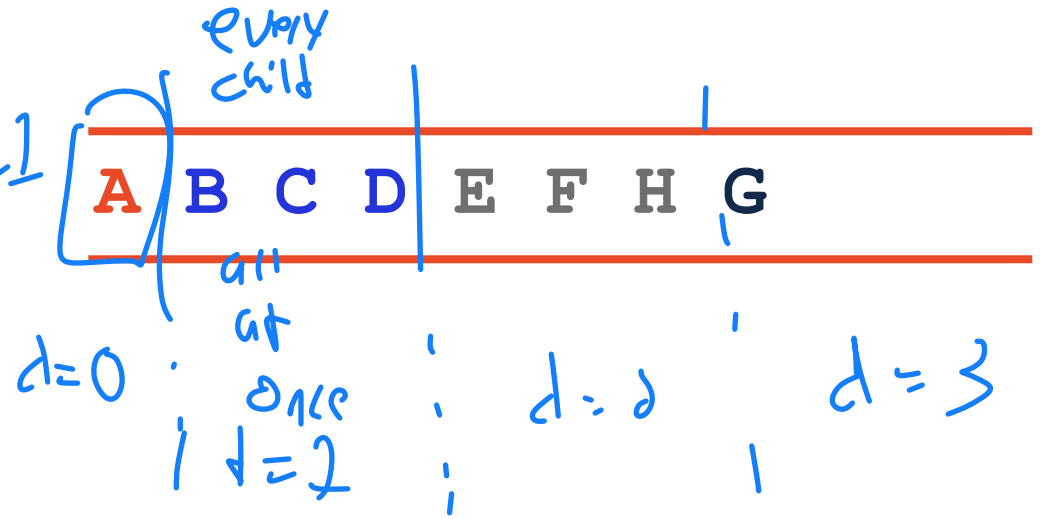
$\sum deg(v) = 2|E|$ in undirected graph

Efficiency: DFS vs BFS

Storage costs tell important story!

BFS: $O(1+m)$

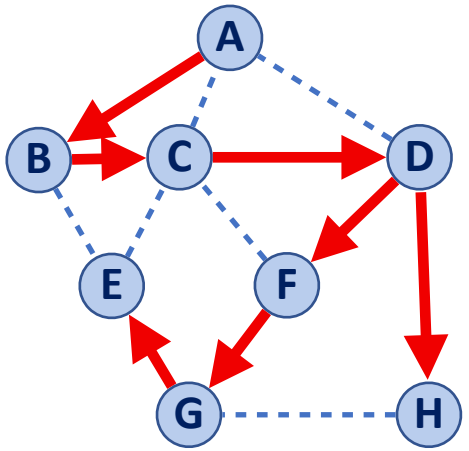




Graph width is size of largest depth vertices

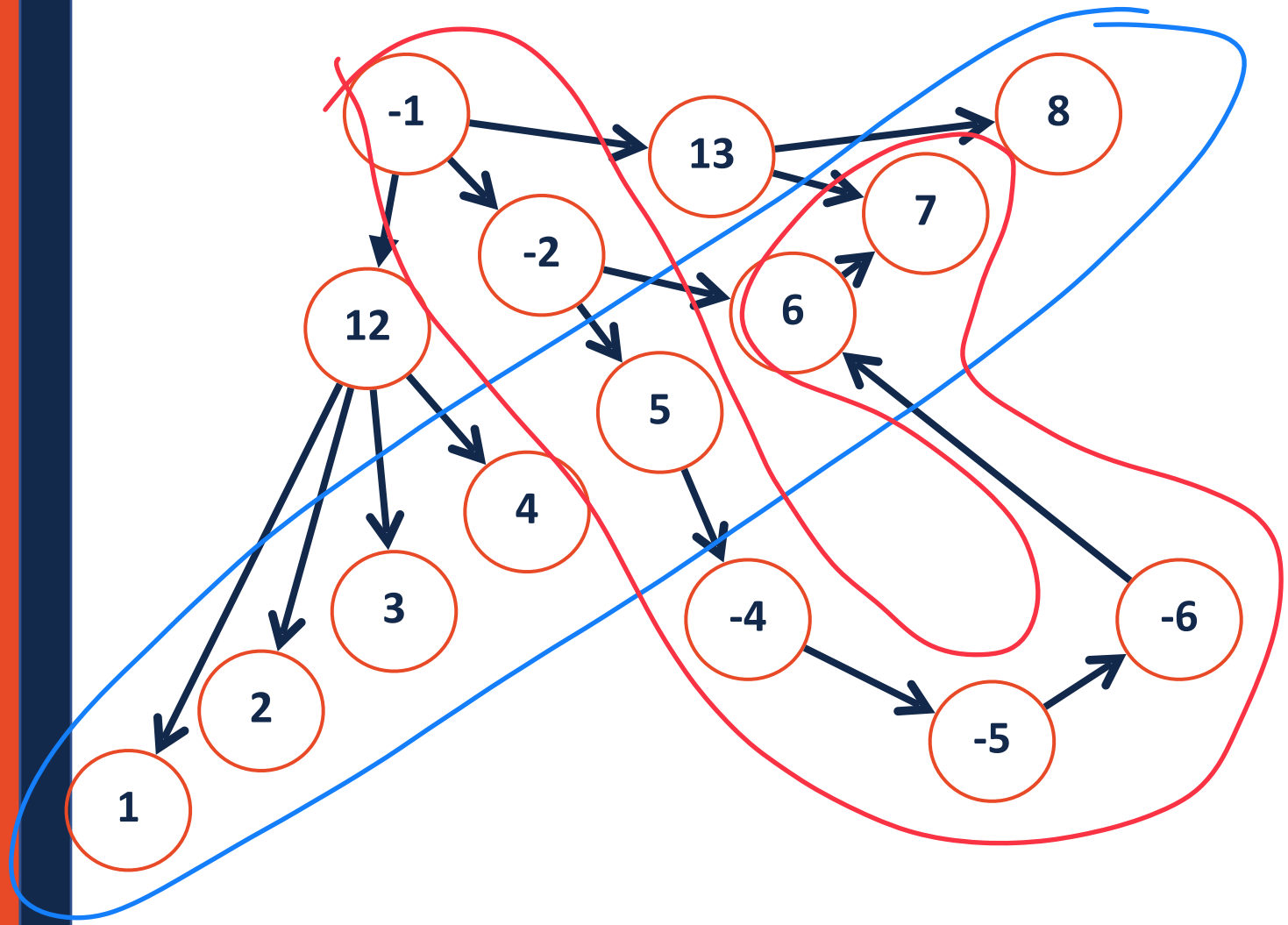
depth == dist

DFS: $O(1+m)$



The stack has worst case of longest path

Space Efficiency: DFS vs BFS



BFS edge width

DFS longest path

Summary: DFS and BFS

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

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

BFS → we always explore all children

Solves unweighted MST

Solves shortest path ✓

Solves cycle detection

Memory bounded by width

DFS → we pick one child to move down

Solves unweighted MST

Solves cycle detection

Memory bounded by longest path

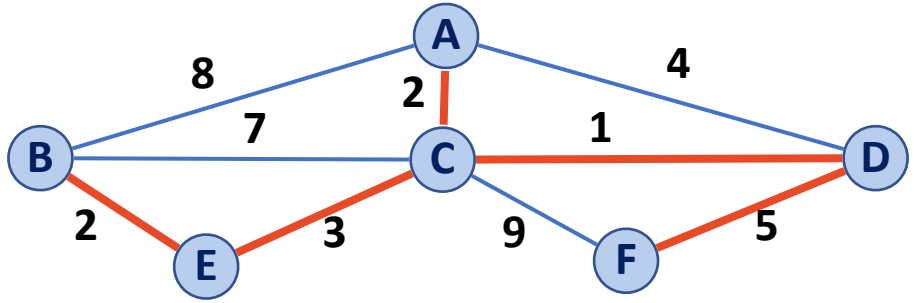
Future work: Depth limit DFS!

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



Graph Algorithms: Minimum Spanning Trees

Vojtěch Jarník (1897-1970)

First wrote algorithm for MST

Robert Prim (1921 — 2021)

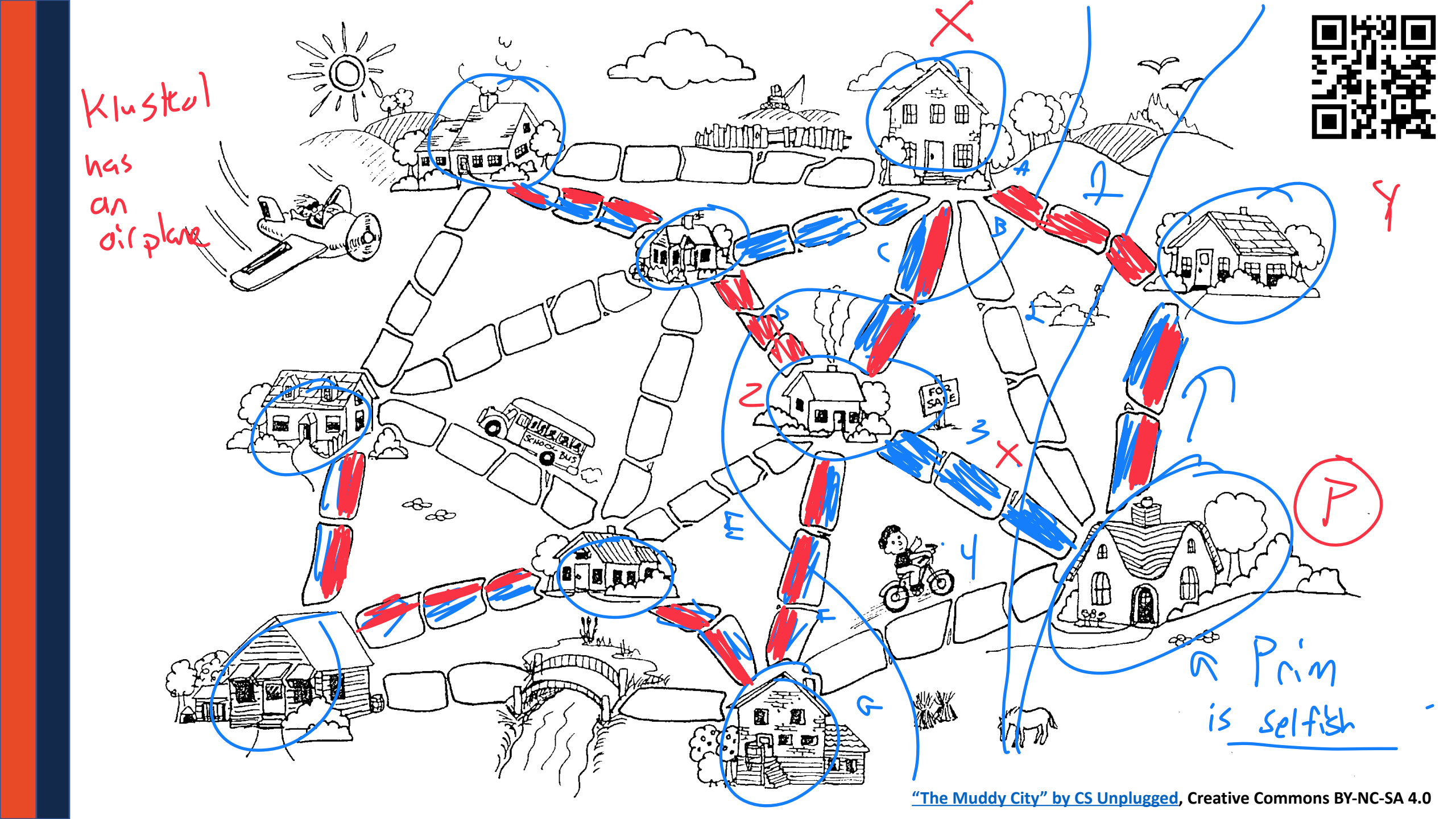
Rediscovered Jarnik's algorithm

(It's named after Prim now though)

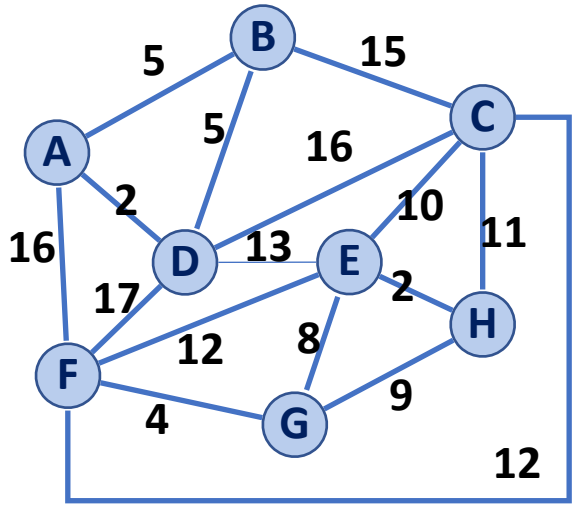
Joseph Kruskal (1928 — 2010)

Worked in Bell Labs with Prim

Came up with his own MST algorithm



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



What information do I need to get efficiently?

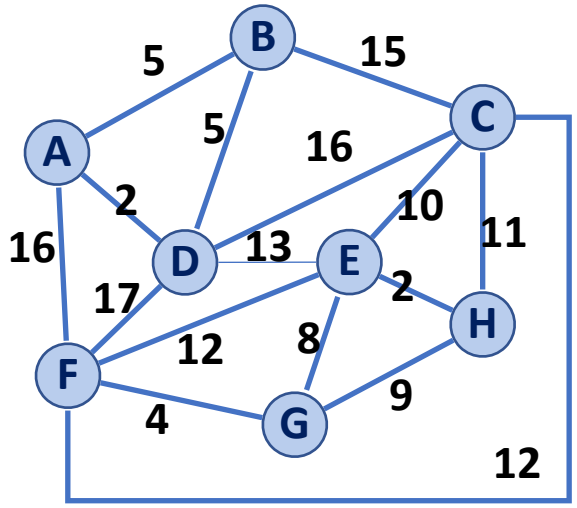
1) The global minimum edge weight 10%

2) Edge connections for a given vertex 70%

3) If two vertices are already connected 30%

4) A visited list of vertices 60%

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



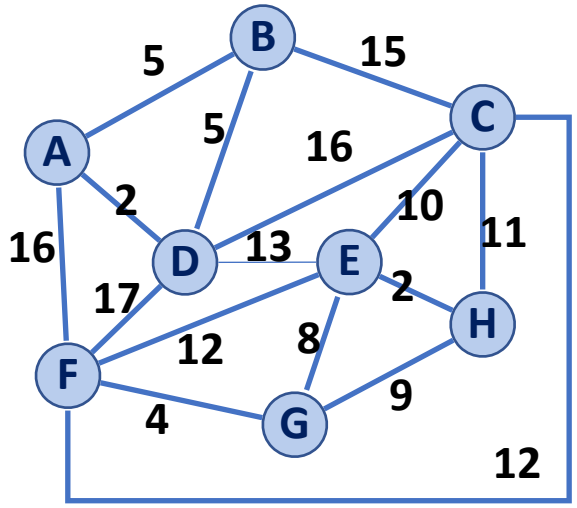
What information do I need to get efficiently?

1) The global minimum edge weight

3) If two vertices are already connected

What does this tell me about my implementation choices?

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)

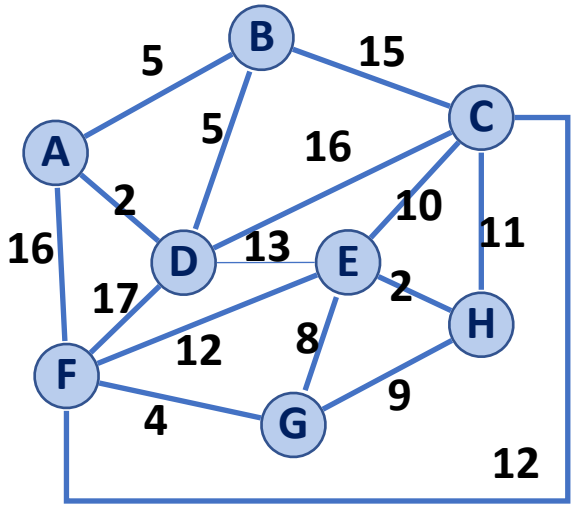
2) We need a **disjoint set** of vertices

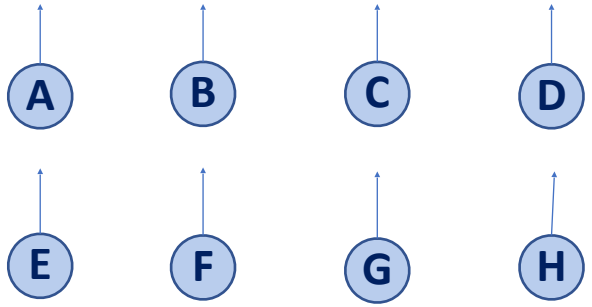
Kruskal's Algorithm

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

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

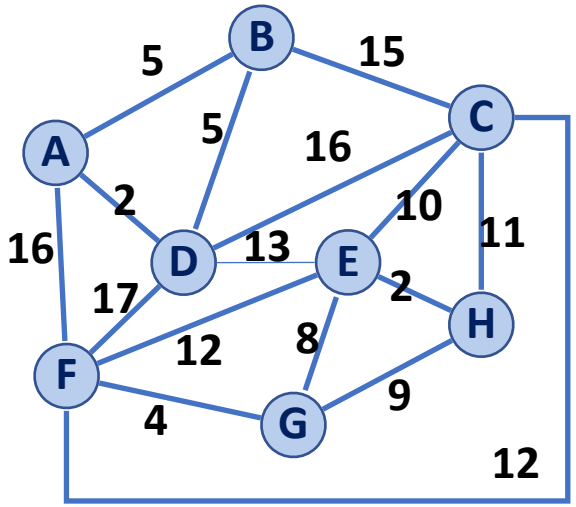
(A, D)
(E, H)
(F, G)
(A, B)
(B, D)
(G, E)
(G, H)
(E, C)
(C, H)
(E, F)
(F, C)
(D, E)
(B, C)
(C, D)
(A, F)
(D, F)

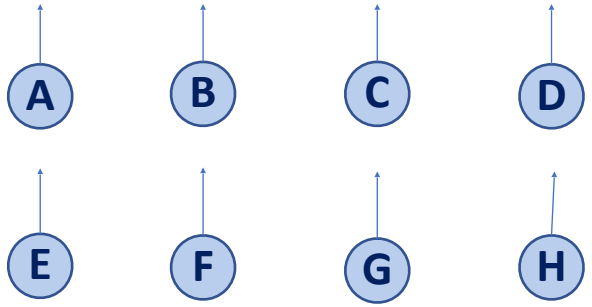




Kruskal's Algorithm

(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

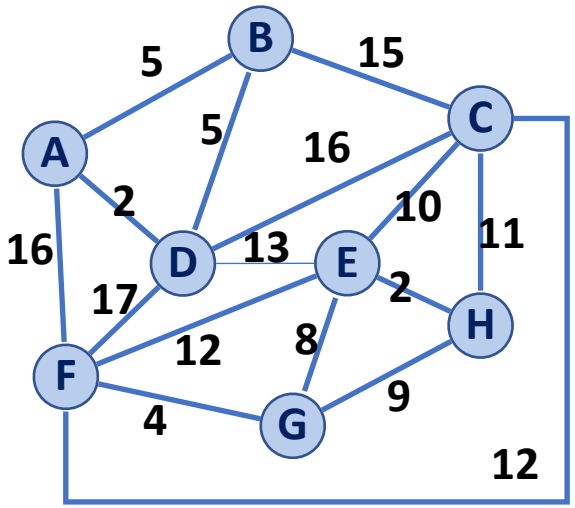
4) Stop when:

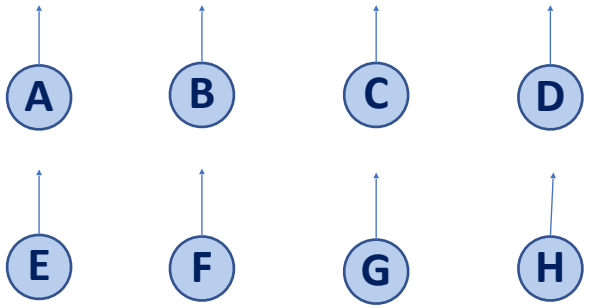
N-1 edges recorded

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)






```
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

```
 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
```

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

What is the Big O?

Kruskal's Algorithm

```
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
```

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

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	

```
 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	O(n)+O(m) + O(m log m)
Sorted Array	O(n) + O(m log m) + O(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
```

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

Bounds of n & m relationship?

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

Connected graph *Complete graph*