CSE 373

NOVEMBER 29TH – MINUMUM SPANNING TREES

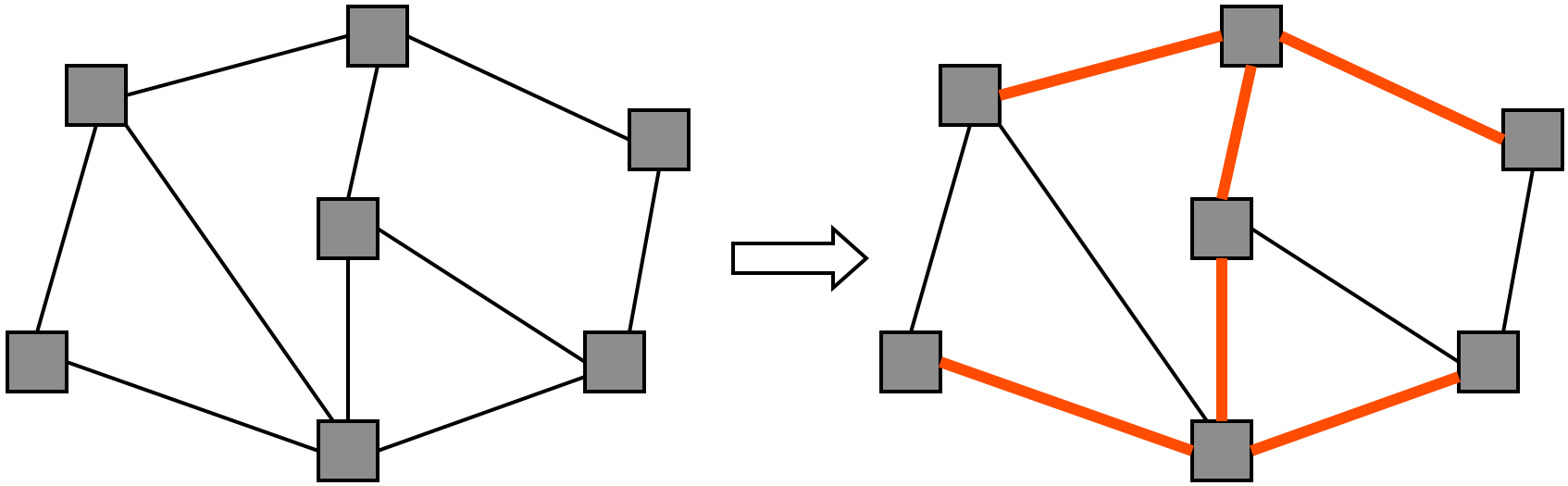
ASSORTED MINUTIAE

- **Project 3**
  - Preliminary part 2 grades out
  - Official grades for part 2 and part 1 regrades (with email) by Friday
- **Written Assignment**
  - Out tonight
  - Sorting and graphs
  - No late days allowed

PROBLEM STATEMENT

Given a *connected* undirected graph G=(V,E), find a minimal subset of edges such that G is still connected

- A graph **G2**=(**V**,**E2**) such that **G2** is connected and removing any edge from **E2** makes **G2** disconnected



OBSERVATIONS

1. **Problem not defined if original graph not connected. Therefore, we know |E| >= |V|-1**

2. **Any solution to this problem is a tree**
   - Recall a tree does not need a root; just means acyclic
   - For any cycle, could remove an edge and still be connected
3. **A tree with |V| nodes has |V|-1 edges**
   - So every solution to the spanning tree problem has **|V|-1** edges

TWO APPROACHES

Different algorithmic approaches to the spanning-tree problem:

1. **Do a graph traversal (e.g., depth-first search, but any traversal will do), keeping track of edges that form a tree**

2. **Iterate through edges; add to output any edge that does not create a cycle**

SECOND APPROACH

Iterate through edges; output any edge that does not create a cycle

Correctness (hand-wavy):

- Goal is to build an acyclic connected graph
- When we add an edge, it adds a vertex to the tree
- The graph is connected, so we reach all vertices

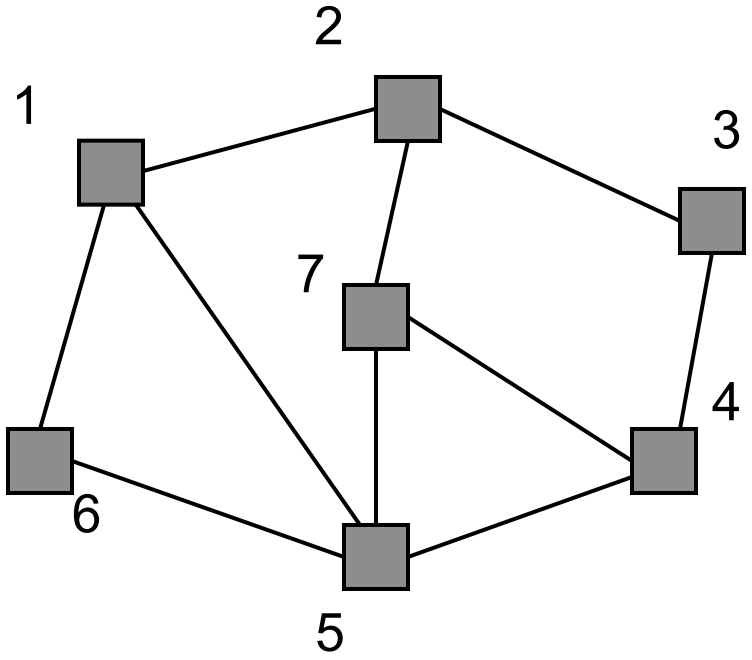
Efficiency:

- Depends on how quickly you can detect cycles
- Reconsider after the example

EXAMPLE

Edges in some arbitrary order:

(1,2), (3,4), (5,6), (5,7),(1,5), (1,6), (2,7), (2,3), (4,5), (4,7)

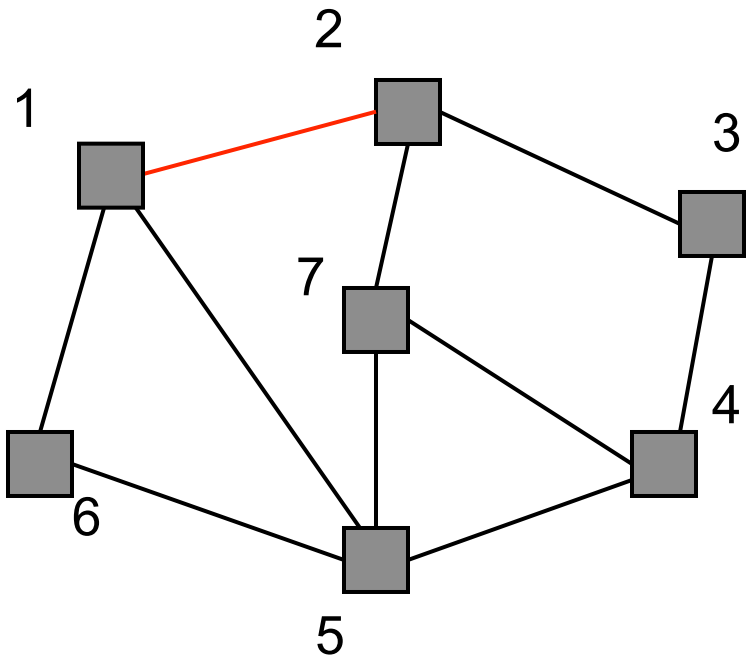


Output:

EXAMPLE

Edges in some arbitrary order:

(1,2), (3,4), (5,6), (5,7),(1,5), (1,6), (2,7), (2,3), (4,5), (4,7)

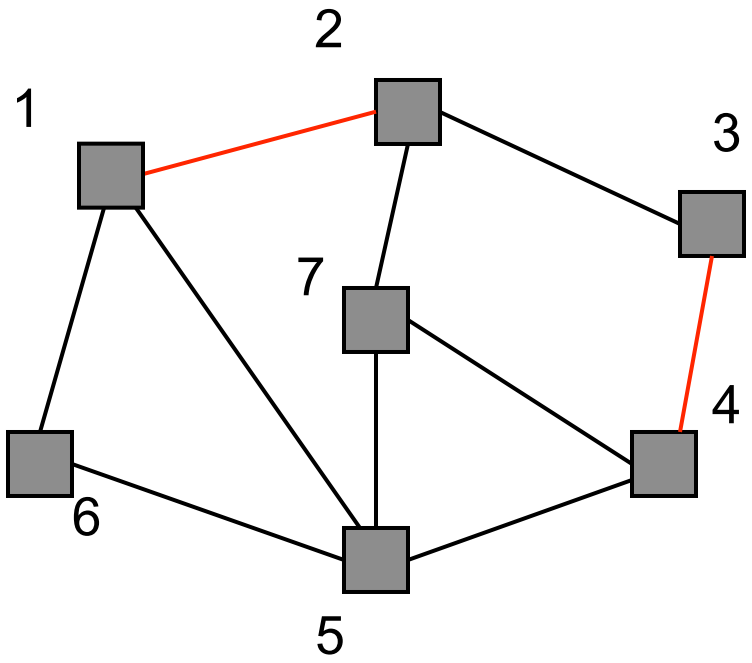


Output: (1,2)

EXAMPLE

Edges in some arbitrary order:

(1,2), (3,4), (5,6), (5,7),(1,5), (1,6), (2,7), (2,3), (4,5), (4,7)

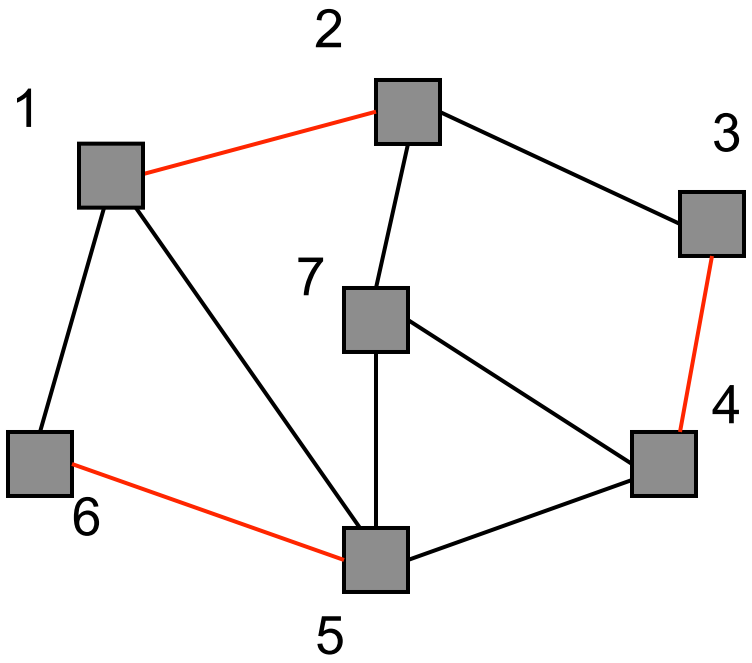


Output: (1,2), (3,4)

EXAMPLE

Edges in some arbitrary order:

(1,2), (3,4), (5,6), (5,7),(1,5), (1,6), (2,7), (2,3), (4,5), (4,7)

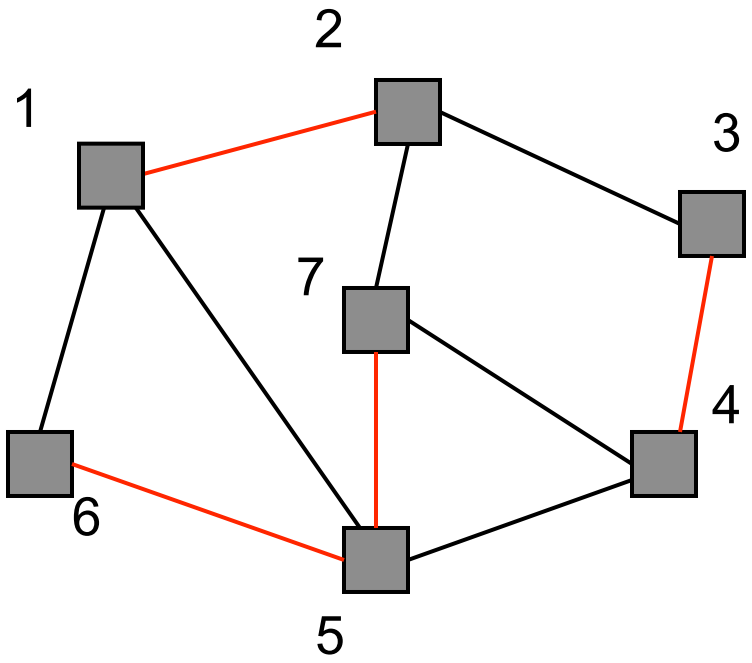


Output: (1,2), (3,4), (5,6),

EXAMPLE

Edges in some arbitrary order:

(1,2), (3,4), (5,6), (5,7),(1,5), (1,6), (2,7), (2,3), (4,5), (4,7)

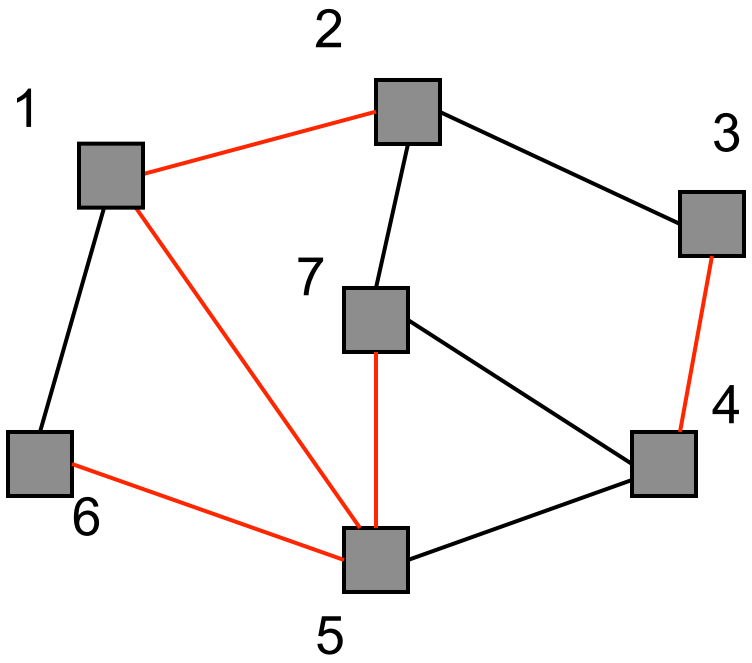


Output: (1,2), (3,4), (5,6), (5,7)

EXAMPLE

Edges in some arbitrary order:

(1,2), (3,4), (5,6), (5,7), (1,5), (1,6), (2,7), (2,3), (4,5), (4,7)

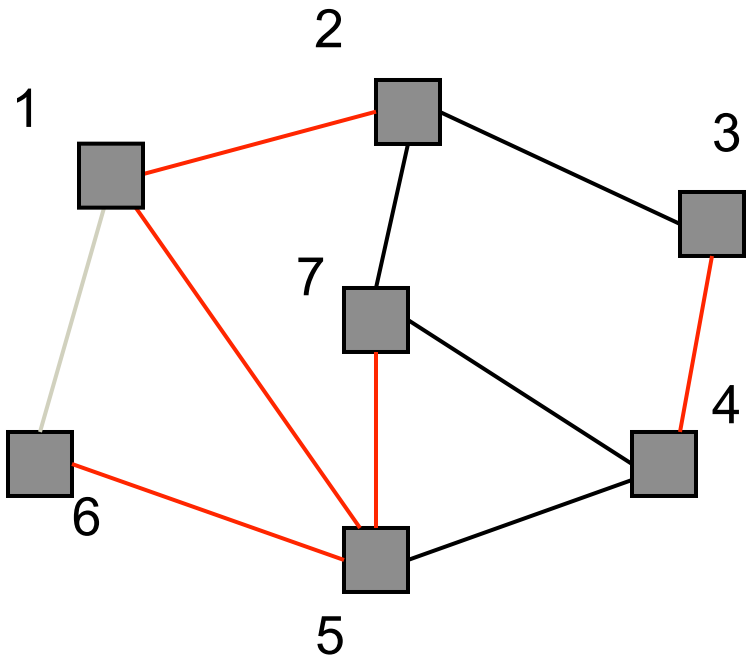


Output: (1,2), (3,4), (5,6), (5,7), (1,5)

EXAMPLE

Edges in some arbitrary order:

(1,2), (3,4), (5,6), (5,7), (1,5), (1,6), **(2,7), (2,3), (4,5), (4,7)**

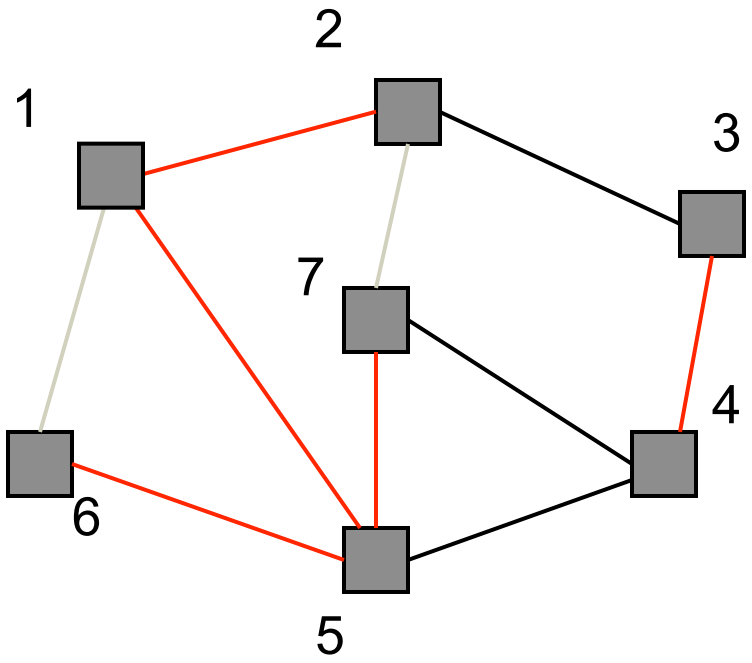


Output: (1,2), (3,4), (5,6), (5,7), (1,5)

EXAMPLE

Edges in some arbitrary order:

(1,2), (3,4), (5,6), (5,7), (1,5), (1,6), (2,7), **(2,3), (4,5), (4,7)**

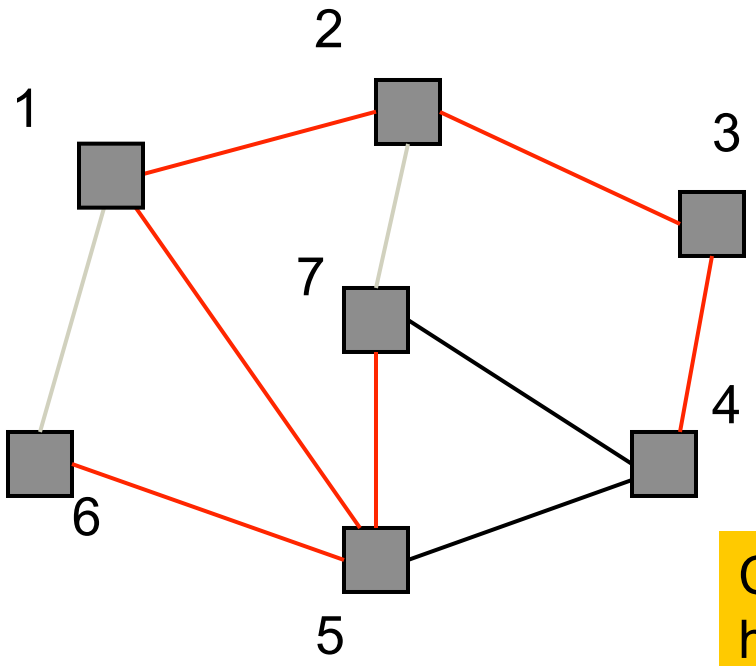


Output: (1,2), (3,4), (5,6), (5,7), (1,5)

EXAMPLE

Edges in some arbitrary order:

(1,2), (3,4), (5,6), (5,7), (1,5), (1,6), (2,7), (2,3), **(4,5), (4,7)**



Can stop once we have |V|-1 edges

Output: (1,2), (3,4), (5,6), (5,7), (1,5), (2,3)

CYCLE DETECTION

To decide if an edge could form a cycle is $O(|V|)$ because we may need to traverse all edges already in the output

So overall algorithm would be $O(|V||E|)$

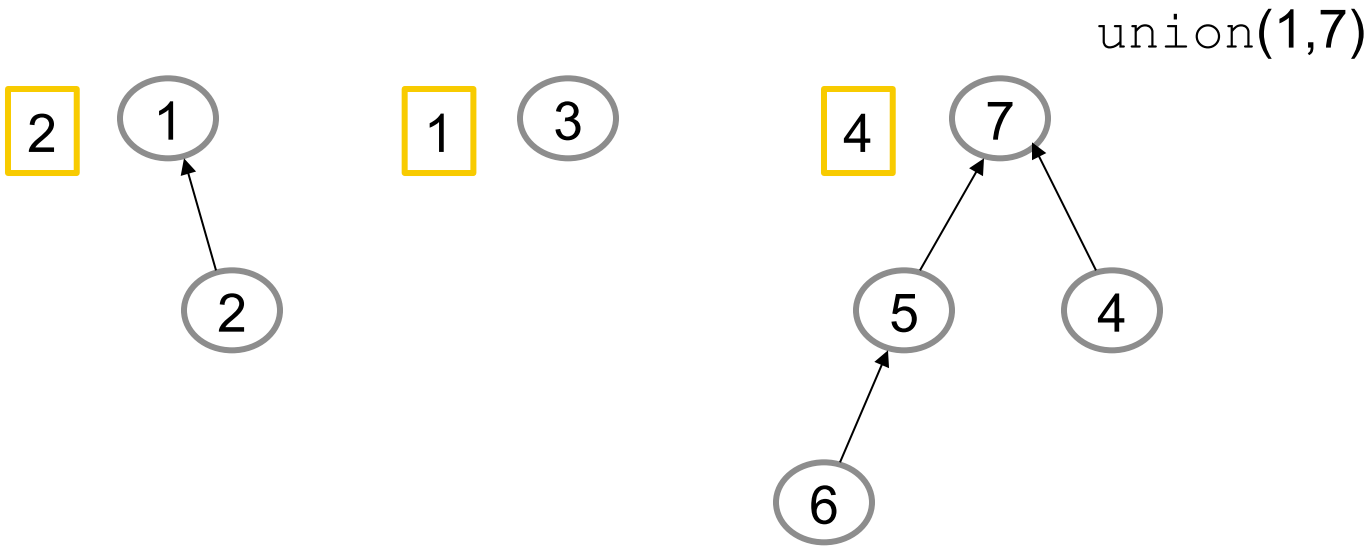
But there is a faster way: union-find!

- Data structure which stores connected sub-graphs
- As we add more edges to the spanning tree, those sub-graphs are joined

WEIGHTED UNION

Weighted union:

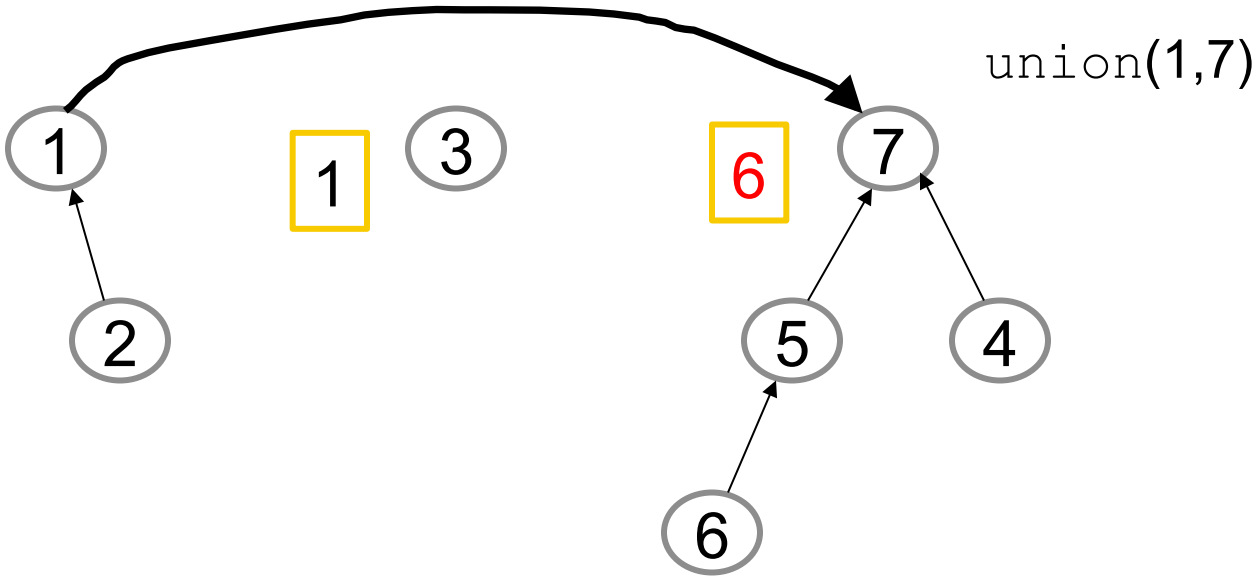
- Always point the *smaller* (total # of nodes) tree to the root of the larger tree



WEIGHTED UNION

Weighted union:

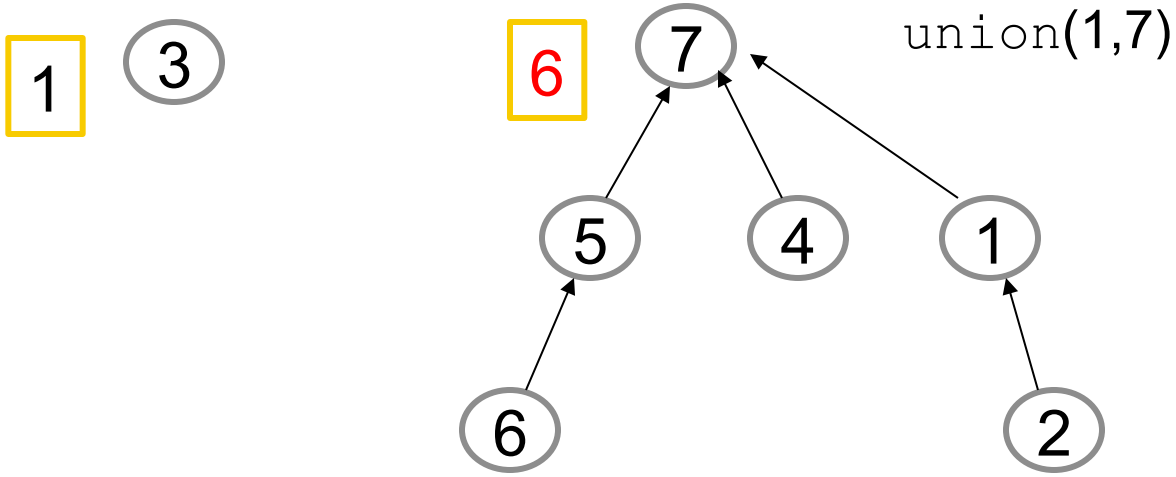
- Always point the *smaller* (total # of nodes) tree to the root of the larger tree
- What just happened to the height of the larger tree?



WEIGHTED UNION

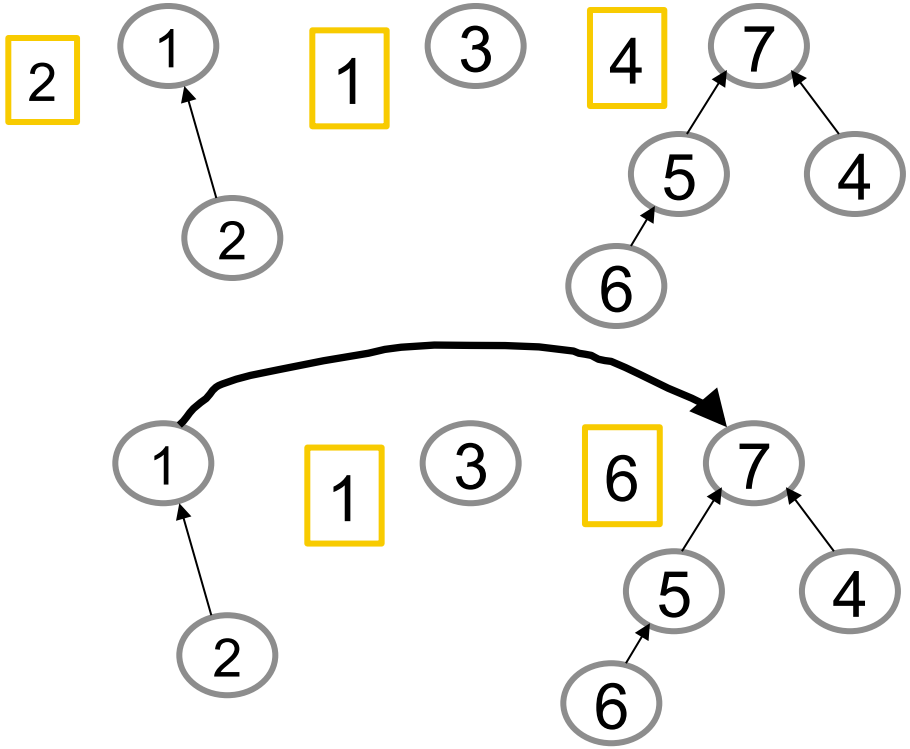
Weighted union:

- Like balancing on an AVL tree, we're trying to keep the traversal from leaf to overall root short



ARRAY IMPLEMENTATION

Keep the *weight* (number of nodes in a second array). Or have one array of objects with two fields. Could keep track of *height*, but that's harder. *Weight* gives us an approximation.



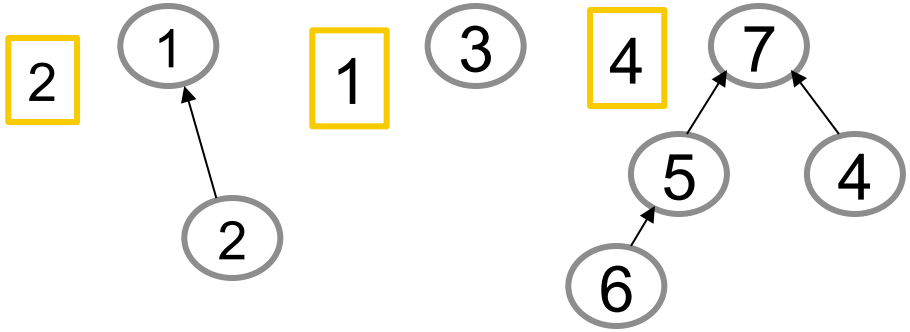
	1	2	3	4	5	6	7
parent	0	1	0	7	7	5	0
weight	2		1				4

	1	2	3	4	5	6	7
parent	7	1	0	7	7	5	0
weight	2		1				6

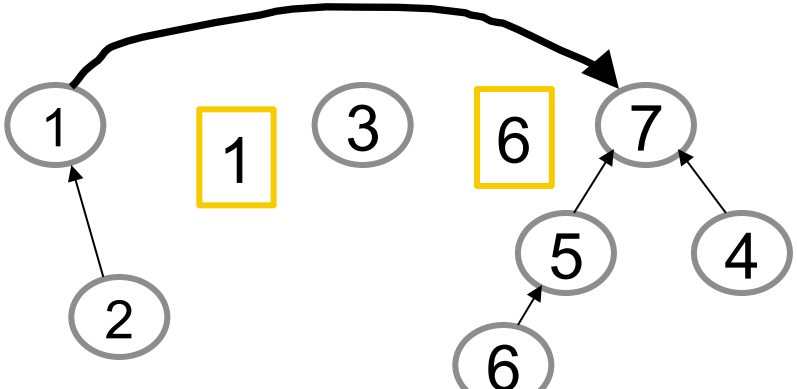
NIFTY TRICK

Actually we do not need a second array…

- Instead of storing 0 for a root, store negation of weight. So parent value < 0 means a root.



	1	2	3	4	5	6	7
parent or weight	-2	1	-1	7	7	5	-4

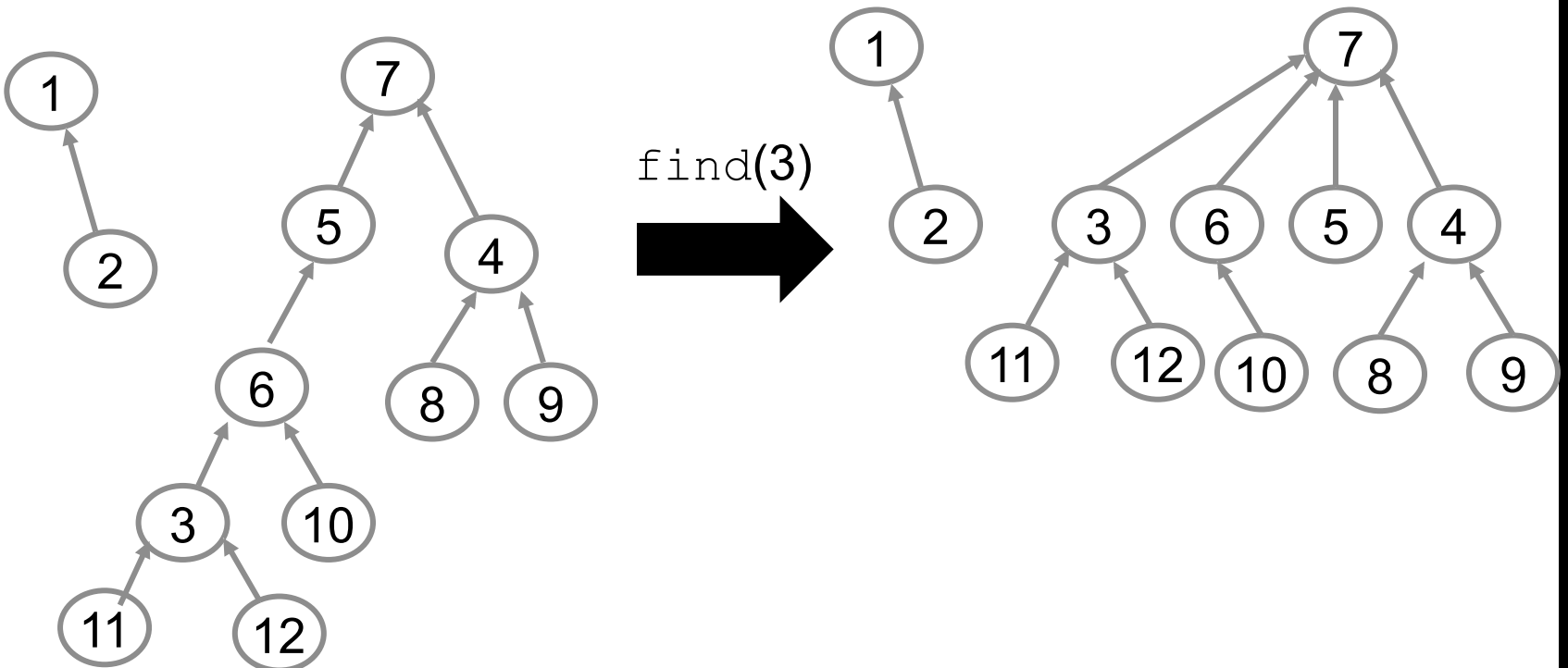


	1	2	3	4	5	6	7
parent or weight	7	1	-1	7	7	5	-6

PATH COMPRESSION

Simple idea: As part of a `find`, change each encountered node's parent to point directly to root

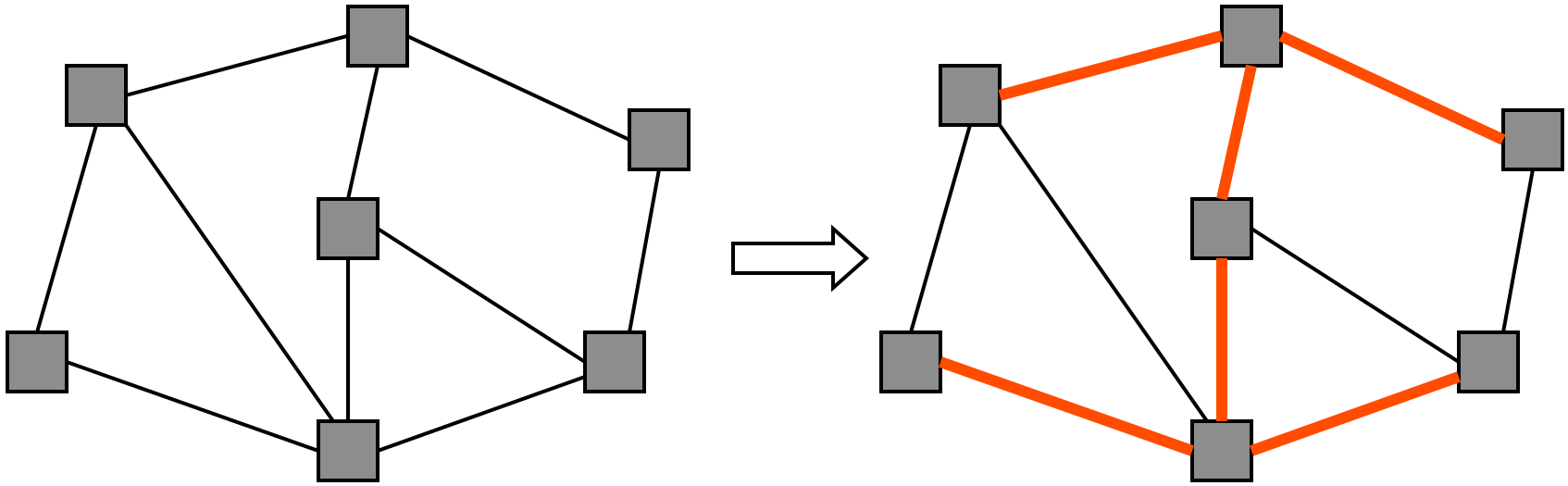
- Faster future **`find`**s for everything on the path (and their descendants)



SPANNING TREES

Given a *connected* undirected graph G=(V,E), find a subset of edges such that G is still connected

- A graph **G2**=(**V**,**E2**) such that **G2** is connected and removing any edge from **E2** makes **G2** disconnected



MINIMAL SPANNING TREES

- **How do we get a minimal spanning tree from a traversal?**

MINIMAL SPANNING TREES

- **How do we get a minimal spanning tree from a traversal?**
  - What parts of a traversal can we change?

MINIMAL SPANNING TREES

- **How do we get a minimal spanning tree from a traversal?**
  - What parts of a traversal can we change?
  - Select which vertex we visit next by which is closest to an old vertex

PRIM'S ALGORITHM

- **A traversal**

PRIM'S ALGORITHM

- **A traversal**
  - Pick a start node

PRIM'S ALGORITHM

- **A traversal**
  - Pick a start node
  - Keep track of all of the vertices you can reach

PRIM'S ALGORITHM

- **A traversal**
  - Pick a start node
  - Keep track of all of the vertices you can reach
  - Add the vertex that is closest (has the edge with smallest weight) to the current spanning tree.

PRIM'S ALGORITHM

- **A traversal**
  - Pick a start node
  - Keep track of all of the vertices you can reach
  - Add the vertex that is closest (has the edge with smallest weight) to the current spanning tree.
- **Is this similar to something we've seen before?**

PRIM'S ALGORITHM

- **Modify Dijkstra's algorithm**

PRIM'S ALGORITHM

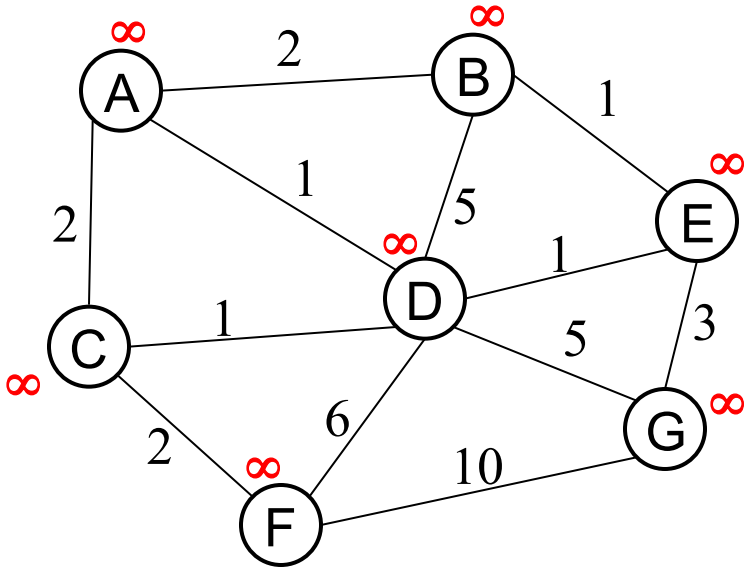
- **Modify Dijkstra's algorithm**
  - Instead of measuring the total length from start to the new vertex, now we only care about the edge from our current spanning tree to new nodes

THE ALGORITHM

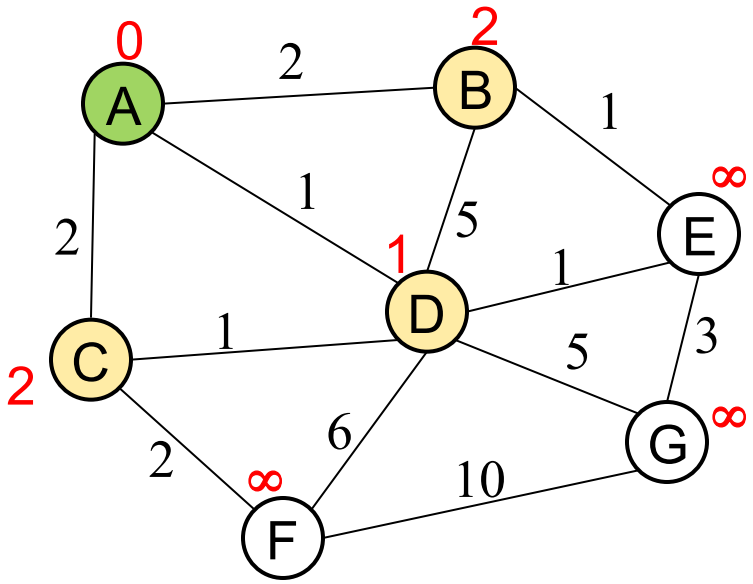
1. **For each node v, set v.cost = ∞ and v.known = false**
2. **Choose any node v**
   a) Mark **v** as known
   b) For each edge **(v,u)** with weight **w**, set **u.cost=w** and **u.prev=v**
3. **While there are unknown nodes in the graph**
   a) Select the unknown node **v** with lowest cost
   b) Mark **v** as known and add **(v, v.prev)** to output
   c) For each edge **(v,u)** with weight **w**,

```
if(w < u.cost) {
    u.cost = w;
u.prev = v;
}
```

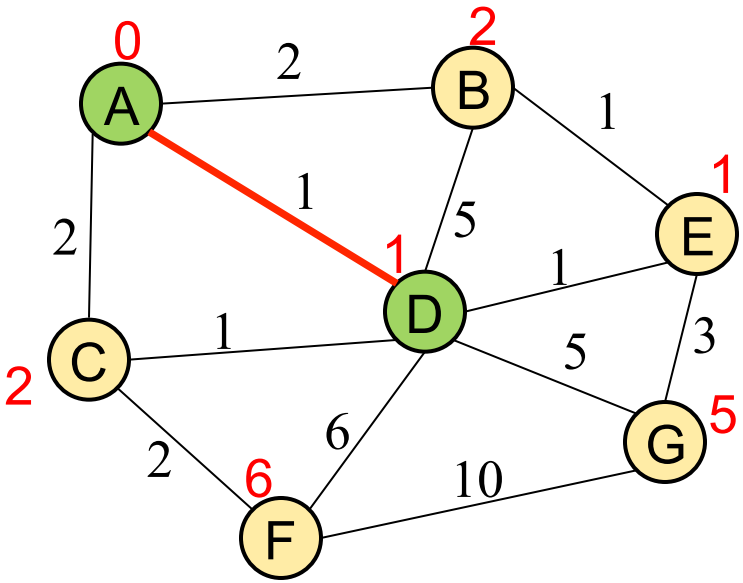
EXAMPLE



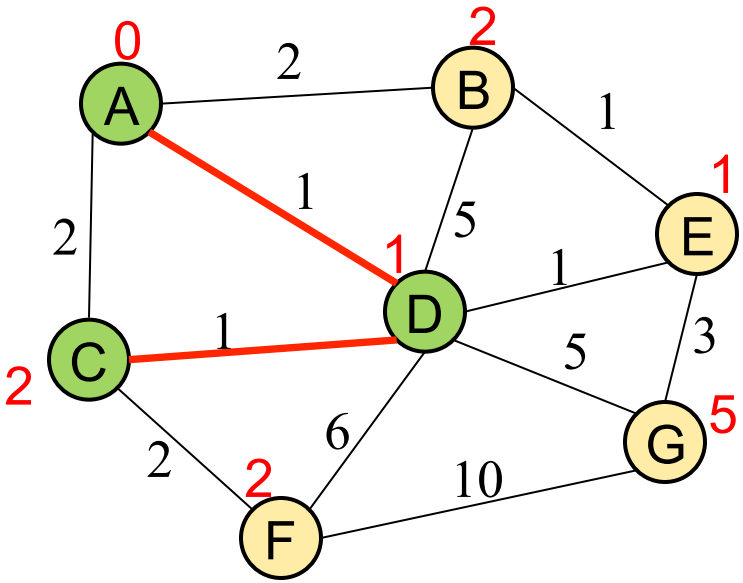
vertex	known?	cost	prev
A		∞	
B		∞	
C		∞	
D		∞	
E		∞	
F		∞	
G		∞	



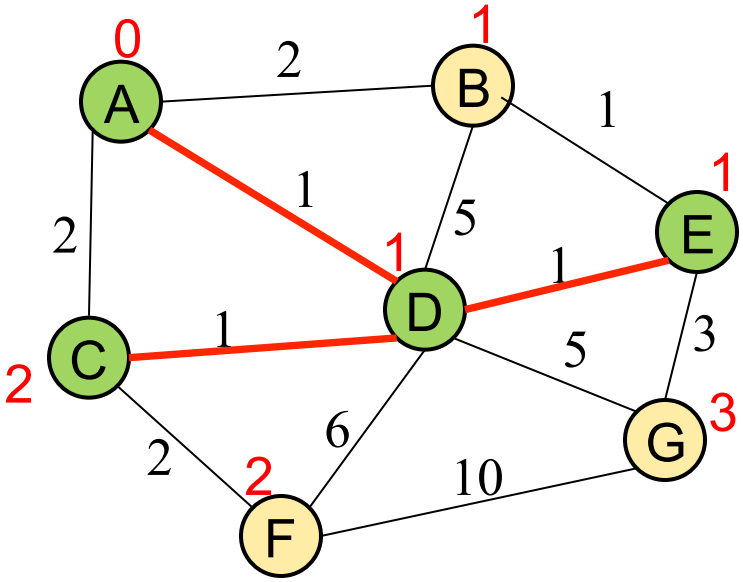
vertex	known?	cost	prev
A	Y	0	
B		2	A
C		2	A
D		1	A
E		∞	
F		∞	
G		∞	



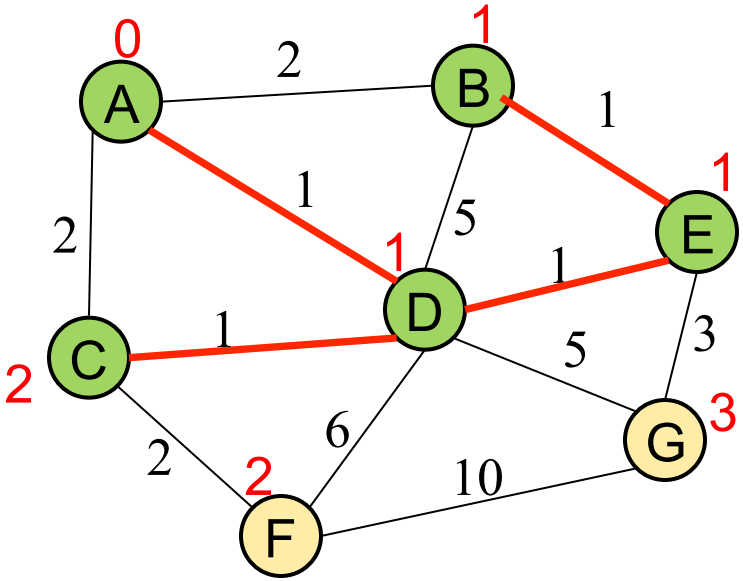
vertex	known?	cost	prev
A	Y	0	
B		2	A
C		1	D
D	Y	1	A
E		1	D
F		6	D
G		5	D



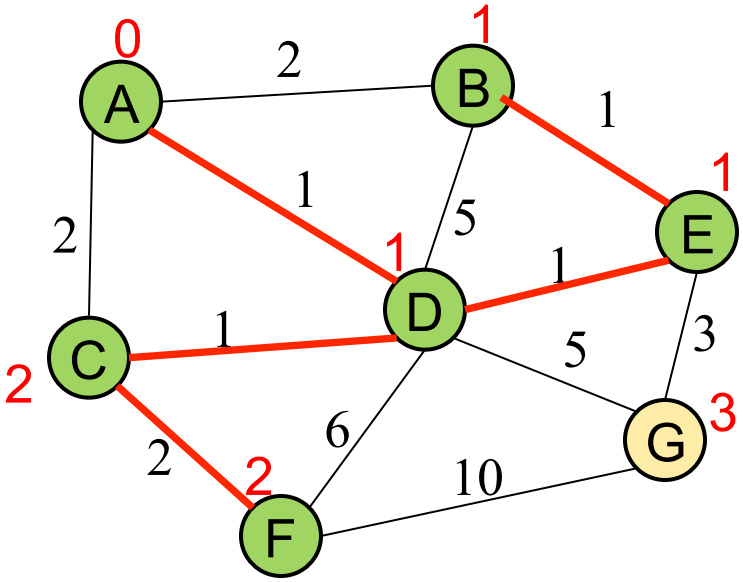
vertex	known?	cost	prev
A	Y	0	
B		2	A
C	Y	1	D
D	Y	1	A
E		1	D
F		2	C
G		5	D



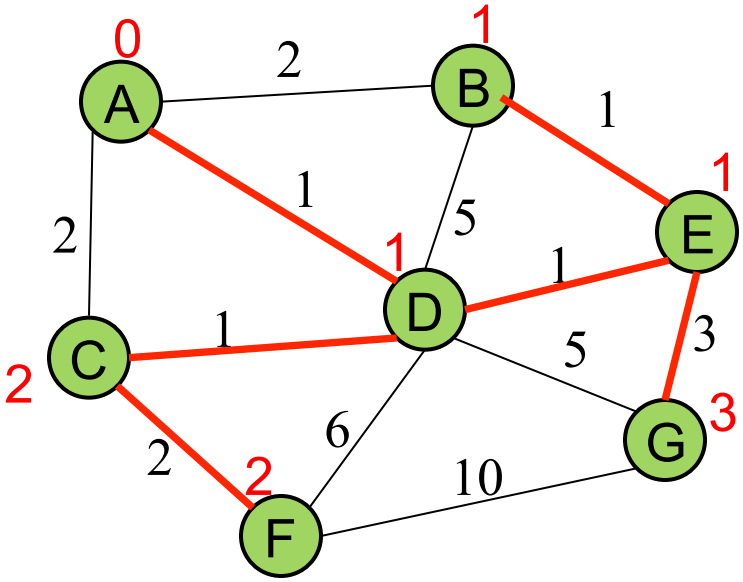
vertex	known?	cost	prev
A	Y	0	
B		1	E
C	Y	1	D
D	Y	1	A
E	Y	1	D
F		2	C
G		3	E



vertex	known?	cost	prev
A	Y	0	
B	Y	1	E
C	Y	1	D
D	Y	1	A
E	Y	1	D
F		2	C
G		3	E



vertex	known?	cost	prev
A	Y	0	
B	Y	1	E
C	Y	1	D
D	Y	1	A
E	Y	1	D
F	Y	2	C
G		3	E



vertex	known?	cost	prev
A	Y	0	
B	Y	1	E
C	Y	1	D
D	Y	1	A
E	Y	1	D
F	Y	2	C
G	Y	3	E

PRIM'S ALGORITHM

- **Does this give us the correct solution? Why?**

PRIM'S ALGORITHM

- **Does this give us the correct solution? Why?**
  - If we consider the "known" cloud as a single vertex, we will never add edges that form a cycle

PRIM'S ALGORITHM

- **Does this give us the correct solution? Why?**
  - If we consider the "known" cloud as a single vertex, we will never add edges that form a cycle
  - Each time, we take the edge that has minimal weight going out of the vertex.

PRIM'S ALGORITHM

- **Does this give us the correct solution? Why?**
  - If we consider the "known" cloud as a single vertex, we will never add edges that form a cycle
  - Each time, we take the edge that has minimal weight going out of the vertex.
  - This is the cheapest way of connecting the two subgraphs.

PRIM'S ALGORITHM

- **What is the runtime?**

PRIM'S ALGORITHM

- **What is the runtime?**
  - Traversals go through all of the edges, in the worst case

PRIM'S ALGORITHM

- **What is the runtime?**
  - Traversals go through all of the edges, in the worst case
    - Need to check if an edge forms a cycle or if it has minimal weight.

PRIM'S ALGORITHM

- **What is the runtime?**
  - Traversals go through all of the edges, in the worst case
    - Need to check if an edge forms a cycle or if it has minimal weight.
    - We can check if it forms a cycle by verifying if the other vertex is in the "known cloud"

PRIM'S ALGORITHM

- **What is the runtime?**
  - Traversals go through all of the edges, in the worst case
    - Need to check if an edge forms a cycle or if it has minimal weight.
    - We can check if it forms a cycle by verifying if the other vertex is in the "known cloud" **O(1)**

PRIM'S ALGORITHM

- **What is the runtime?**
  - Traversals go through all of the edges, in the worst case
    - Need to check if an edge forms a cycle or if it has minimal weight.
    - We can check if it forms a cycle by verifying if the other vertex is in the "known cloud" **O(1)**
    - How long to check if it is minimal?

PRIM'S ALGORITHM

- **What is the runtime?**
  - Traversals go through all of the edges, in the worst case
    - Need to check if an edge forms a cycle or if it has minimal weight.
    - We can check if it forms a cycle by verifying if the other vertex is in the "known cloud" **O(1)**
    - How long to check if it is minimal? **O(log |V|)** if we use a priority queue

PRIM'S ALGORITHM

- **O(|E| log |V|)**
  - We can use a priority queue to store all of our vertices, and let the edges to them be the priority.

PRIM'S ALGORITHM

- **O(|E| log |V|)**
  - We can use a priority queue to store all of our vertices, and let the edges to them be the priority.
  - Use the decreaseKey() function when the edge to a vertex changes.

PRIM'S ALGORITHM

- **O(|E| log |V|)**
  - We can use a priority queue to store all of our vertices, and let the edges to them be the priority.
  - Use the decreaseKey() function when the edge to a vertex changes.
  - Without the priority queue, both Prim's and Dijkstra's run in **O(|E||V|)**

KRUSKAL'S ALGORITHM

- **Prim's algorithm works from the vertices, and builds a contiguous spanning tree**

KRUSKAL'S ALGORITHM

- **Prim's algorithm works from the vertices, and builds a contiguous spanning tree**
  - The spanning tree grows out from a single vertex

KRUSKAL'S ALGORITHM

- **Prim's algorithm works from the vertices, and builds a contiguous spanning tree**
  - The spanning tree grows out from a single vertex
- **Kruskal's Algorithm adds edges based on their weight**

KRUSKAL'S ALGORITHM

- **Prim's algorithm works from the vertices, and builds a contiguous spanning tree**
  - The spanning tree grows out from a single vertex
- **Kruskal's Algorithm adds edges based on their weight**
  - Must check for cycles

KRUSKAL'S ALGORITHM

- **Prim's algorithm works from the vertices, and builds a contiguous spanning tree**
  - The spanning tree grows out from a single vertex
- **Kruskal's Algorithm adds edges based on their weight**
  - Must check for cycles
  - Use the union-find data structure to speed up this operation

KRUSKAL'S ALGORITHM

- **Pseudocode:**

KRUSKAL'S ALGORITHM

- **Pseudocode:**
  - Sort the edges (or place them into a heap)
  - Create a union-find data structure with all separate vertices

KRUSKAL'S ALGORITHM

- **Pseudocode:**
  - Sort the edges (or place them into a heap)
  - Create a union-find data structure with all separate vertices
  - For each edge, add it to the minimum spanning tree if it does not form a cycle

KRUSKAL'S ALGORITHM

- **Pseudocode:**
  - Sort the edges (or place them into a heap)
  - Create a union-find data structure with all separate vertices
  - For each edge, add it to the minimum spanning tree if the two vertices don't have the same representative in the union find

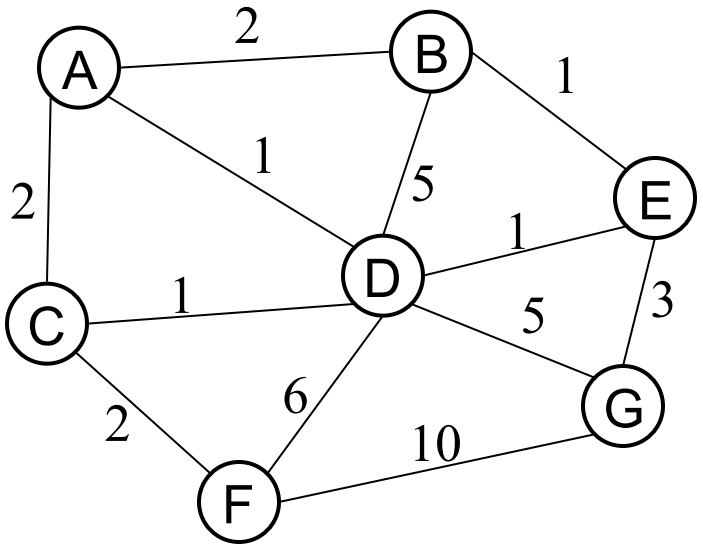
KRUSKAL'S ALGORITHM

- **Pseudocode:**
  - Sort the edges (or place them into a heap)
  - Create a union-find data structure with all separate vertices
  - For each edge, add it to the minimum spanning tree if the two vertices don't have the same representative in the union find
  - Union the two vertices in the union find

KRUSKAL'S ALGORITHM

- **Pseudocode:**
  - Sort the edges (or place them into a heap)
  - Create a union-find data structure with all separate vertices
  - For each edge, add it to the minimum spanning tree if the two vertices don't have the same representative in the union find
  - Union the two vertices in the union find
  - Stop after you've added |V|-1 edges

EXAMPLE



Edges in sorted order:

1: (A,D), (C,D), (B,E), (D,E)

2: (A,B), (C,F), (A,C)

3: (E,G)

5: (D,G), (B,D)

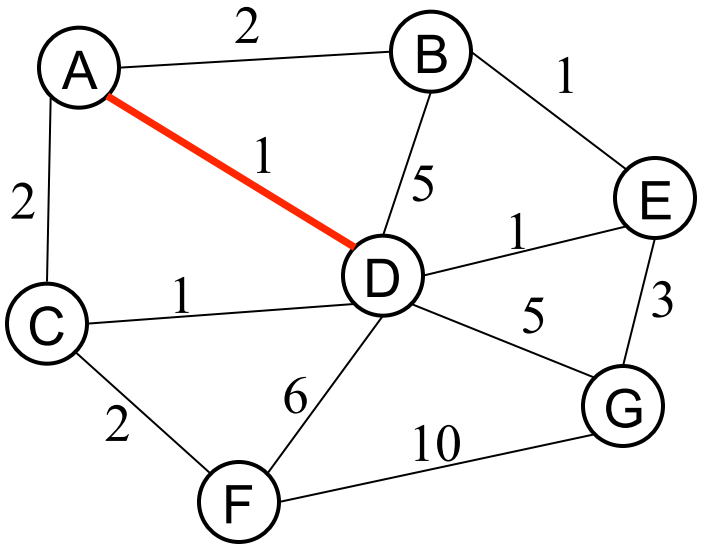
6: (D,F)

10: (F,G)

Output:

Note: At each step, the union/find sets are the trees in the forest

EXAMPLE



Edges in sorted order:

1: (A,D), (C,D), (B,E), (D,E)

2: (A,B), (C,F), (A,C)

3: (E,G)

5: (D,G), (B,D)

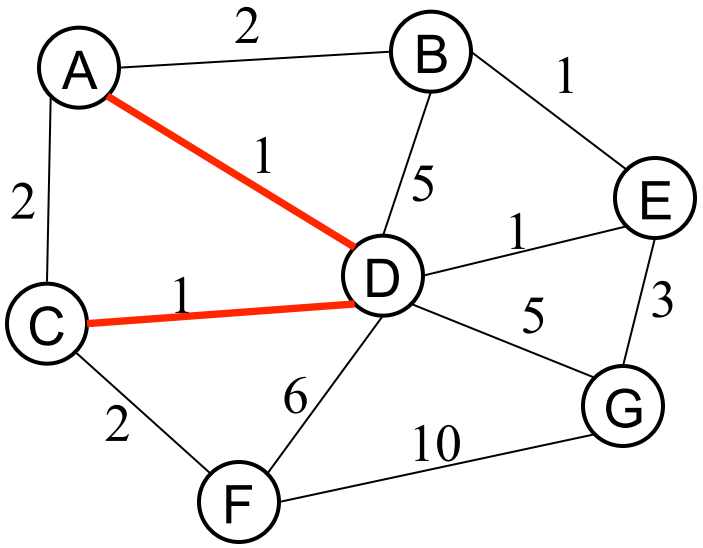
6: (D,F)

10: (F,G)

Output: (A,D)

Note: At each step, the union/find sets are the trees in the forest

EXAMPLE



Edges in sorted order:

1: (A,D), (C,D), (B,E), (D,E)

2: (A,B), (C,F), (A,C)

3: (E,G)

5: (D,G), (B,D)

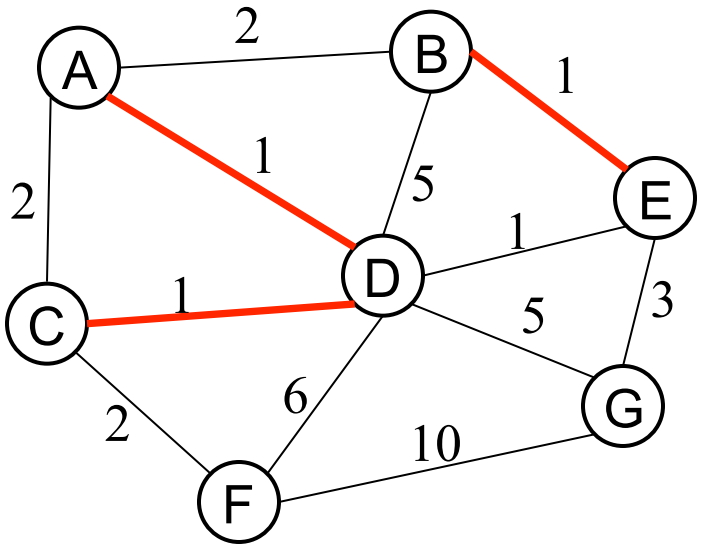
6: (D,F)

10: (F,G)

Output: (A,D), (C,D)

Note: At each step, the union/find sets are the trees in the forest

EXAMPLE



Edges in sorted order:

1: (A,D), (C,D), (B,E), (D,E)

2: (A,B), (C,F), (A,C)

3: (E,G)

5: (D,G), (B,D)

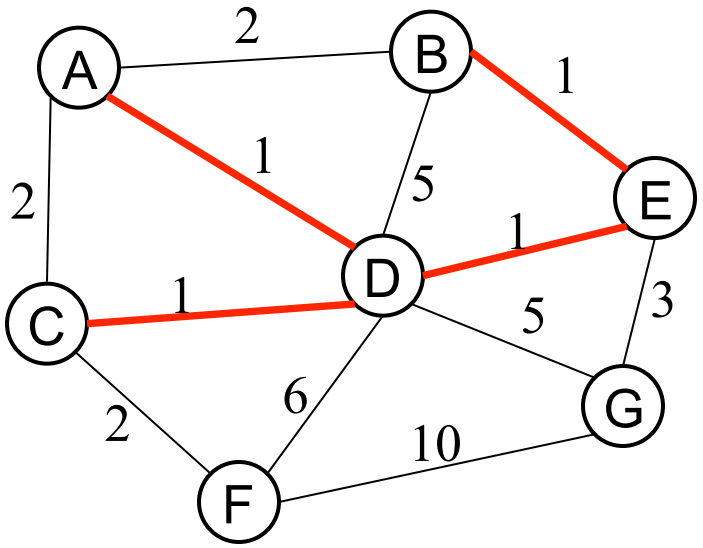
6: (D,F)

10: (F,G)

Output: (A,D), (C,D), (B,E)

Note: At each step, the union/find sets are the trees in the forest

EXAMPLE



Edges in sorted order:

1: (A,D), (C,D), (B,E), (D,E)

2: (A,B), (C,F), (A,C)

3: (E,G)

5: (D,G), (B,D)

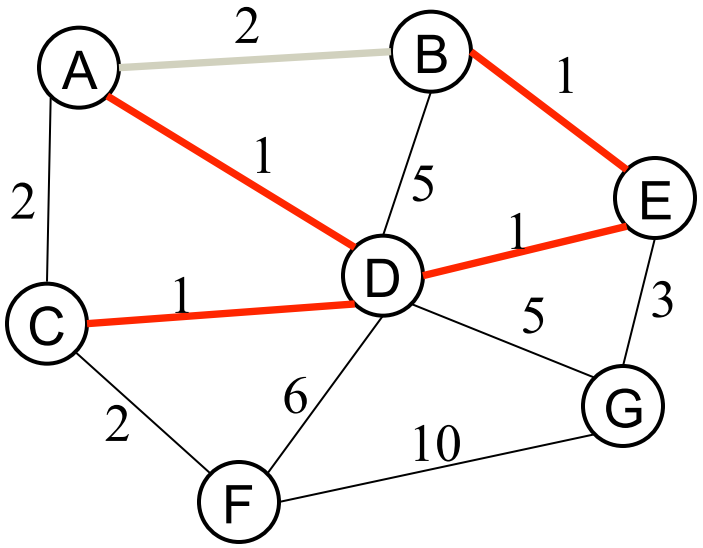
6: (D,F)

10: (F,G)

Output: (A,D), (C,D), (B,E), (D,E)

Note: At each step, the union/find sets are the trees in the forest

EXAMPLE



Edges in sorted order:

1: (A,D), (C,D), (B,E), (D,E)

2: (A,B), **(C,F), (A,C)**

3: **(E,G)**

5: **(D,G), (B,D)**

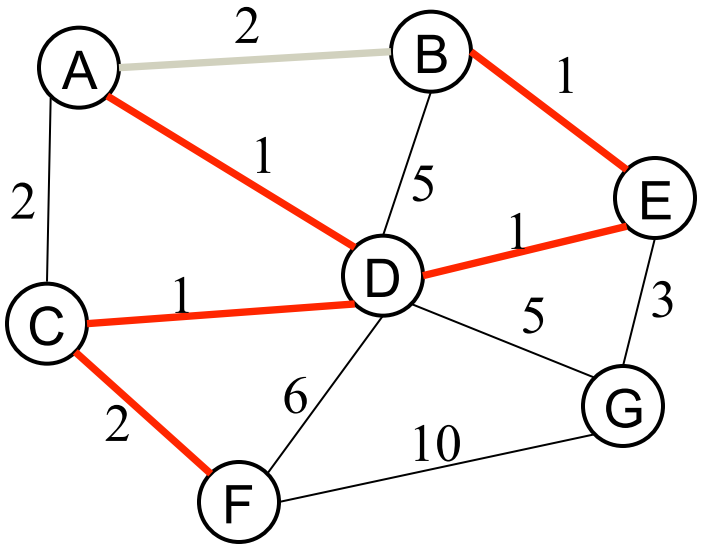
6: **(D,F)**

10: **(F,G)**

Output: (A,D), (C,D), (B,E), (D,E)

Note: At each step, the union/find sets are the trees in the forest

EXAMPLE



Edges in sorted order:

1: (A,D), (C,D), (B,E), (D,E)

2: (A,B), (C,F), **(A,C)**

3: **(E,G)**

5: **(D,G), (B,D)**

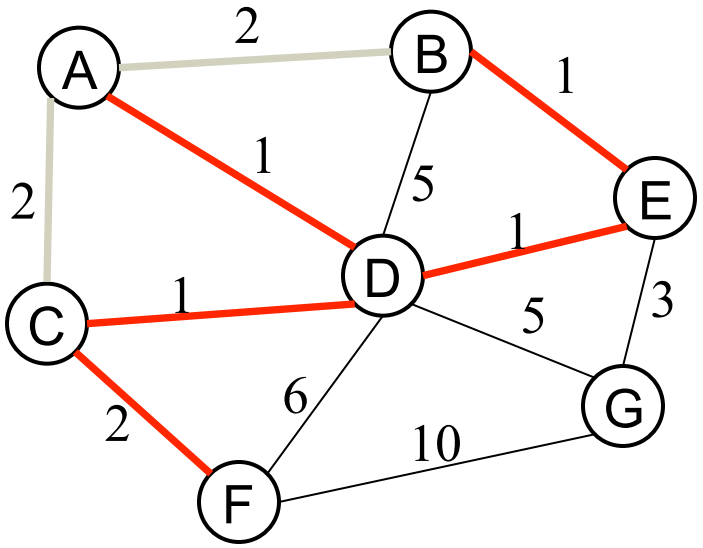
6: **(D,F)**

10: **(F,G)**

Output: (A,D), (C,D), (B,E), (D,E), (C,F)

Note: At each step, the union/find sets are the trees in the forest

EXAMPLE



Edges in sorted order:

1: (A,D), (C,D), (B,E), (D,E)

2: (A,B), (C,F), (A,C)

3: (E,G)

5: (D,G), (B,D)

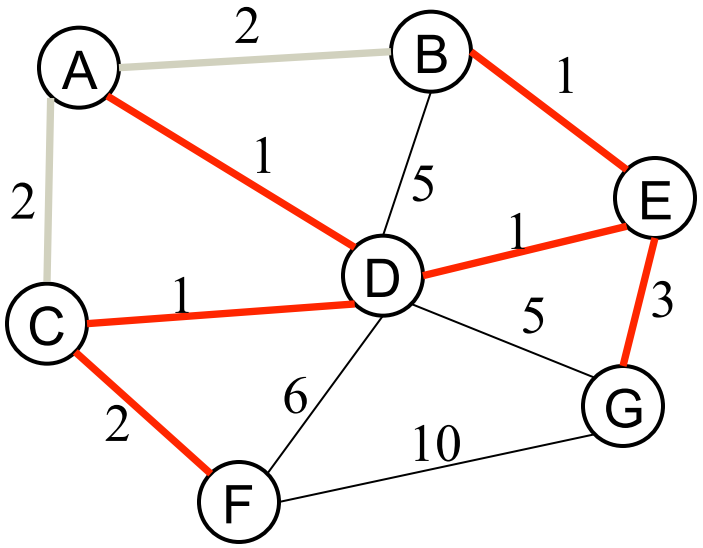
6: (D,F)

10: (F,G)

Output: (A,D), (C,D), (B,E), (D,E), (C,F)

Note: At each step, the union/find sets are the trees in the forest

EXAMPLE



Edges in sorted order:

1: (A,D), (C,D), (B,E), (D,E)

2: (A,B), (C,F), (A,C)

3: (E,G)

5: **(D,G), (B,D)**

6: **(D,F)**

10: **(F,G)**

Output: (A,D), (C,D), (B,E), (D,E), (C,F), (E,G)

Note: At each step, the union/find sets are the trees in the forest

KRUSKAL'S ALGORITHM

- **Runtime**

KRUSKAL'S ALGORITHM

- **Runtime**
  - Put edges into a heap

KRUSKAL'S ALGORITHM

- **Runtime**
  - Put edges into a heap **O(|E|)** Floyd's method!

KRUSKAL'S ALGORITHM

- **Runtime**
  - Put edges into a heap **O(|E|)** Floyd's method!
  - Until the MST is complete:

KRUSKAL'S ALGORITHM

- **Runtime**
  - Put edges into a heap **O(|E|)** Floyd's method!
  - Until the MST is complete:
    - Pull the minimum edge out of the heap

KRUSKAL'S ALGORITHM

- **Runtime**
  - Put edges into a heap **O(|E|)** Floyd's method!
  - Until the MST is complete:
    - Pull the minimum edge out of the heap **O(log |E|)**

KRUSKAL'S ALGORITHM

- **Runtime**
  - Put edges into a heap **O(|E|)** Floyd's method!
  - Until the MST is complete:
    - Pull the minimum edge out of the heap **O(log |E|)**
    - Check if it forms a cycle

KRUSKAL'S ALGORITHM

- **Runtime**
  - Put edges into a heap **O(|E|)** Floyd's method!
  - Until the MST is complete:
    - Pull the minimum edge out of the heap **O(log |E|)**
    - Check if it forms a cycle **O(log |V|)**

KRUSKAL'S ALGORITHM

- **Runtime**
  - Put edges into a heap **O(|E|)** Floyd's method!
  - Until the MST is complete:
    - Pull the minimum edge out of the heap **O(log |E|)**
    - Check if it forms a cycle **O(log |V|)**

KRUSKAL'S ALGORITHM

- **Runtime**
  - Put edges into a heap **O(|E|)** Floyd's method!
  - Until the MST is complete:
    - Pull the minimum edge out of the heap **O(log |E|)**
    - Check if it forms a cycle **O(log |V|)**
  - **How many times does the loop run?**

KRUSKAL'S ALGORITHM

- **Runtime**
  - Put edges into a heap **O(|E|)** Floyd's method!
  - Until the MST is complete:
    - Pull the minimum edge out of the heap **O(log |E|)**
    - Check if it forms a cycle **O(log |V|)**
  - **How many times does the loop run? O(E)**

KRUSKAL'S ALGORITHM

- **Runtime**
  - Put edges into a heap **O(|E|)** Floyd's method!
  - Until the MST is complete:
    - Pull the minimum edge out of the heap **O(log |E|)**
    - Check if it forms a cycle **O(log |V|)**
  - **How many times does the loop run? O(E)**
  - **O(|E| log |E|)**

COMPARISONS

- Prim's
  - O(|E| log |V|)
- Kruskal's
  - O(|E| log |E|)
- Since |E| must be at least |V|-1 for the graph to be connected, which do we prefer?

COMPARISONS

- **Prim's**
  - **O(|E| log |V|)**
- **Kruskal's**
  - **O(|E| log |E|)**
- **Since |E| must be at least |V|-1 for the graph to be connected, which do we prefer?**
  - Since |E| is at most $|V|^2$, log|E| is at most $\log(|V|^2)$ which is 2log|V|.

COMPARISONS

- **Prim's**
  - **O(|E| log |V|)**
- **Kruskal's**
  - **O(|E| log |E|)**
- **Since |E| must be at least |V|-1 for the graph to be connected, which do we prefer?**
  - Since |E| is at most $|V|^2$, log|E| is at most $\log(|V|^2)$ which is 2log|V|.
  - So log|E| is O(log|V|)

CONCLUSIONS

- **Prim's and Kruskal's both run in O(|E| log |V|)**

CONCLUSIONS

- **Prim's and Kruskal's both run in $O(|E| \log |V|)$**
- **An undirected graph has a unique minimum spanning tree if all of its edge weights are unique.**

CONCLUSIONS

- Prim's and Kruskal's both run in $O(|E| \log |V|)$
- An undirected graph has a unique minimum spanning tree if all of its edge weights are unique.
- If graphs have multiple edges of the same weight, it is possible (but not necessary) that there are many spanning trees of the same weight

CONCLUSIONS

- Prim's and Kruskal's both run in $O(|E| \log |V|)$
- An undirected graph has a unique minimum spanning tree if all of its edge weights are unique.
- If graphs have multiple edges of the same weight, it is possible (but not necessary) that there are many spanning trees of the same weight