CSE332: Data Abstractions

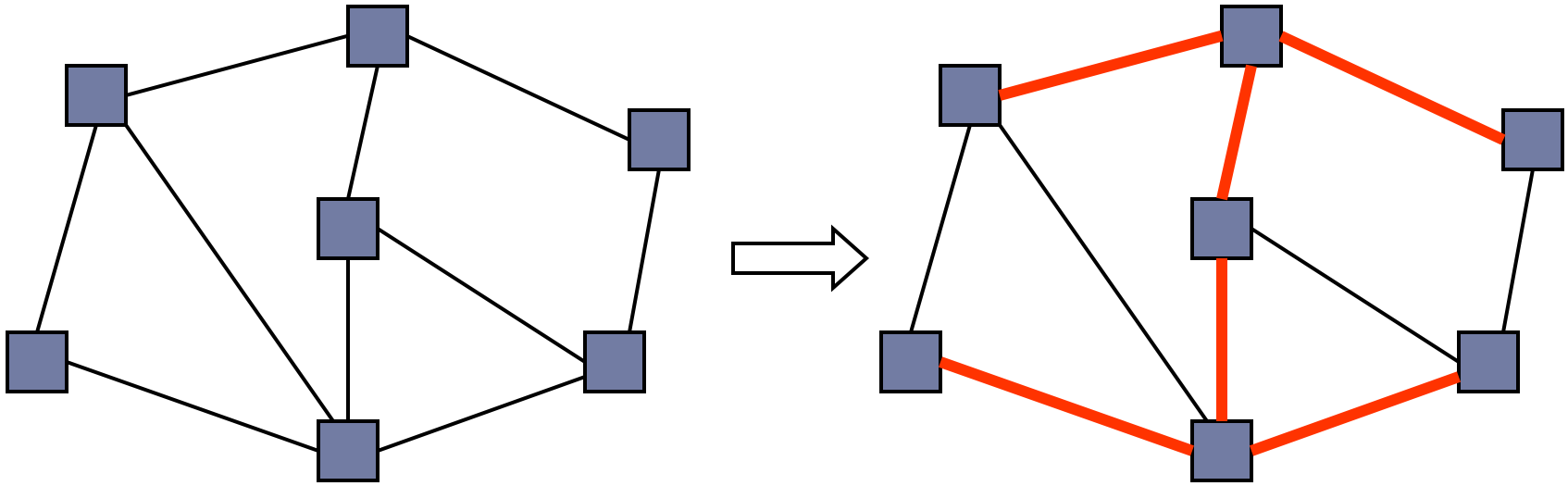
Lecture 18: Minimum Spanning Trees

Tyler Robison

Summer 2010

Spanning trees

- A simple problem: Given a *connected* graph **G**=(**V**,**E**), find a minimal subset of the edges such that the graph is still connected
  - A graph **G2**=(**V**,**E2**) such that **G2** is connected and removing any edge from **E2** makes **G2** disconnected



Observations

1. Any solution to this problem is a tree
   - A tree in a graph sense; no root, just acyclic & connected
   - Why no cycle?
     - For any cycle, could remove an edge and still be connected

2. Solution not unique unless original graph was already a tree

3. Problem ill-defined if original graph not connected

4. A spanning tree with **|V|** nodes has **|V|-1** edges
   - So every solution to the spanning tree problem has **|V|-1** edges

Motivation

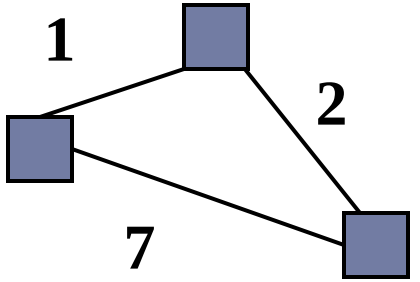
General idea: A spanning tree connects all the nodes with as few edges as possible

Examples:

- Network connectivity
- Power connectivity

In real-world scenarios, we are often concerned with costs/distances:
ex: cost of laying network cable from city A to city B

- Would prefer lower cost



This is the minimum spanning tree problem

Two approaches

Different algorithmic approaches to the minimum spanning-tree problem:

1. Prim's Algorithm: Very similar to Dijkstra's algorithm for shortest path, with a small modification

2. Kruskal's Algorithm: Order all edges in graph by weight; step through each edge and add if it doesn't create a cycle; stop when we have |V|-1 edges

Prim's Algorithm Idea

Idea: Grow a tree by adding an edge from the "known" vertices to the "unknown" vertices.

Pick the edge with the smallest weight that connects "known" to "unknown."

Recall Dijkstra "picked the edge with closest known distance to the source."

- But that's not what we want here
- Here we are interested in distance to 'something in the cloud', not some source vertex
- Otherwise identical

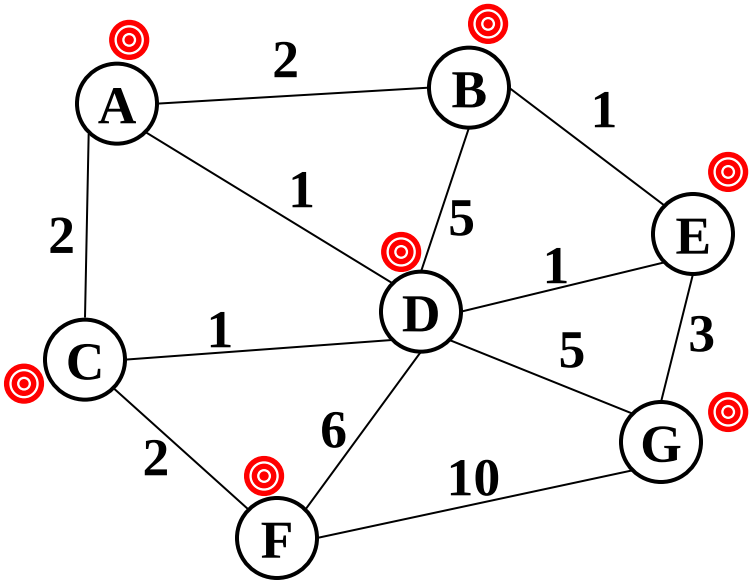
Prim's algorithm

1. For each node **v**, set **v.cost = ∞** and **v.known = false**
2. Choose some starting vertex **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;
}
```

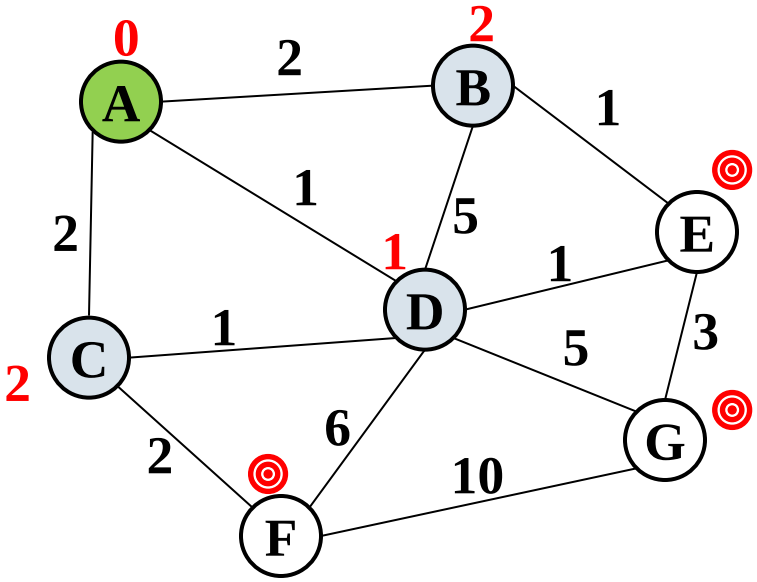
Which part is different from Dijkstra's?

Example



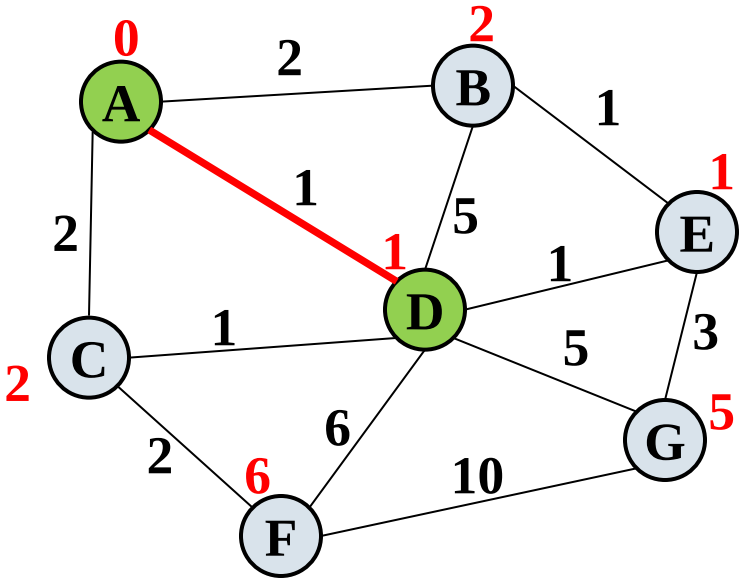
vertex	known?	cost	prev
A		??	
B		??	
C		??	
D		??	
E		??	
F		??	
G		??	

Example



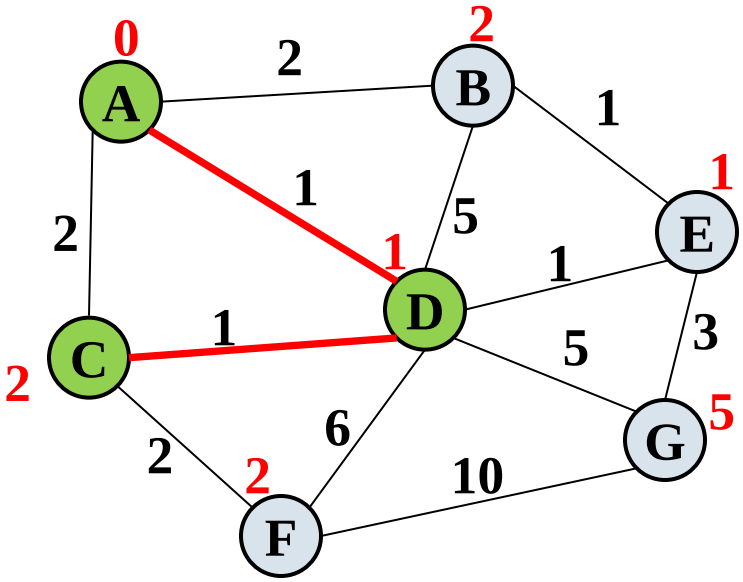
vertex	known?	cost	prev
A	Y	0	
B		2	A
C		2	A
D		1	A
E		??	
F		??	
G		??	

Example



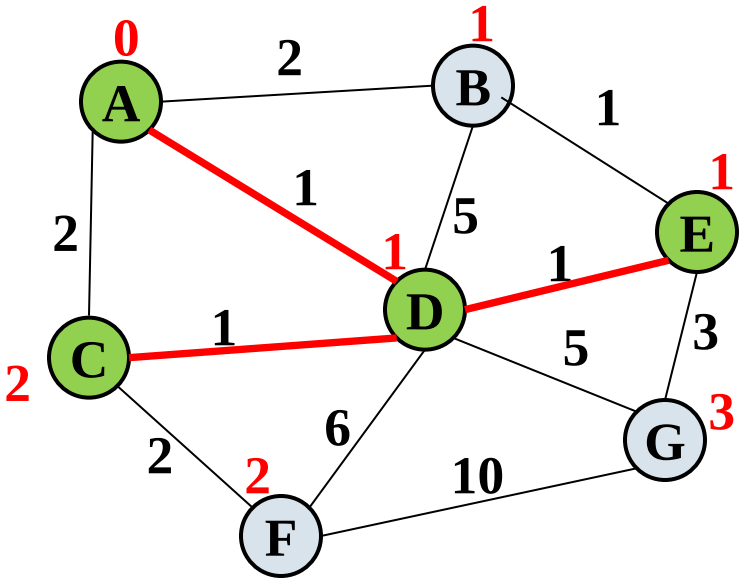
vertex	known?	cost	prev
A	Y	0	
B		2	A
C		1	D
D	Y	1	A
E		1	D
F		6	D
G		5	D

Example



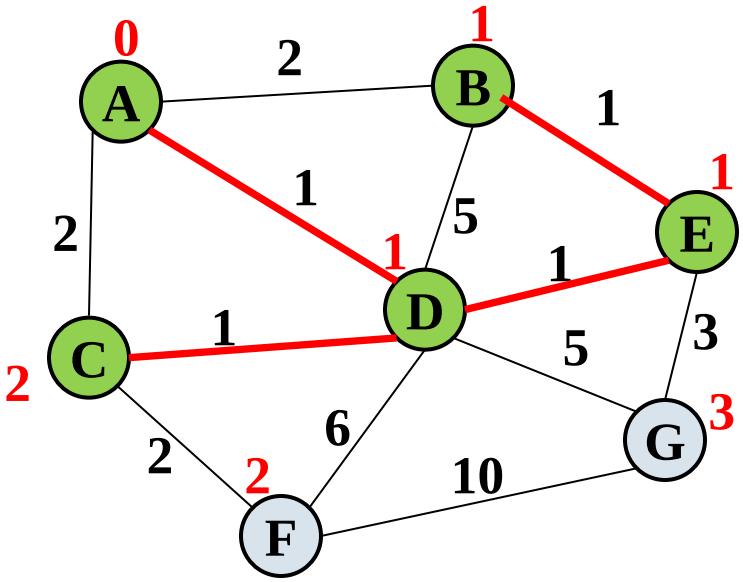
vertex	known?	cost	prev
A	Y	0	
B		2	A
C	Y	1	D
D	Y	1	A
E		1	D
F		2	C
G		5	D

Example



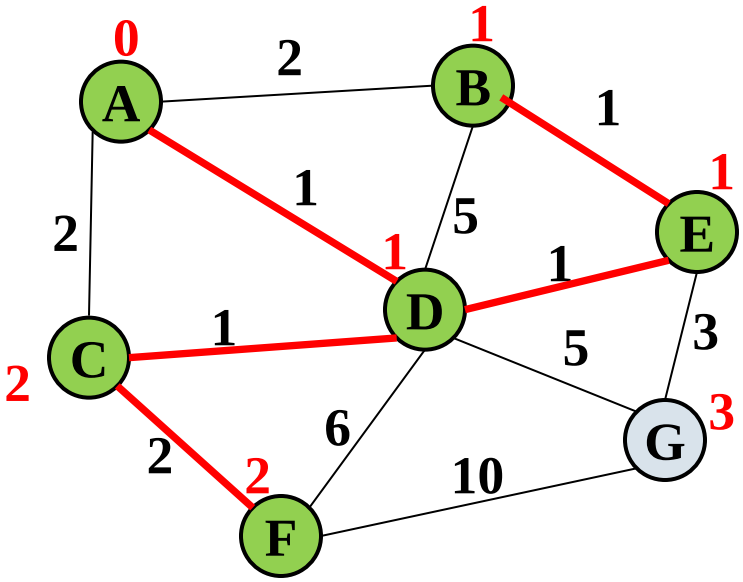
vertex	known?	cost	prev
A	Y	0	
B		1	E
C	Y	1	D
D	Y	1	A
E	Y	1	D
F		2	C
G		3	E

Example



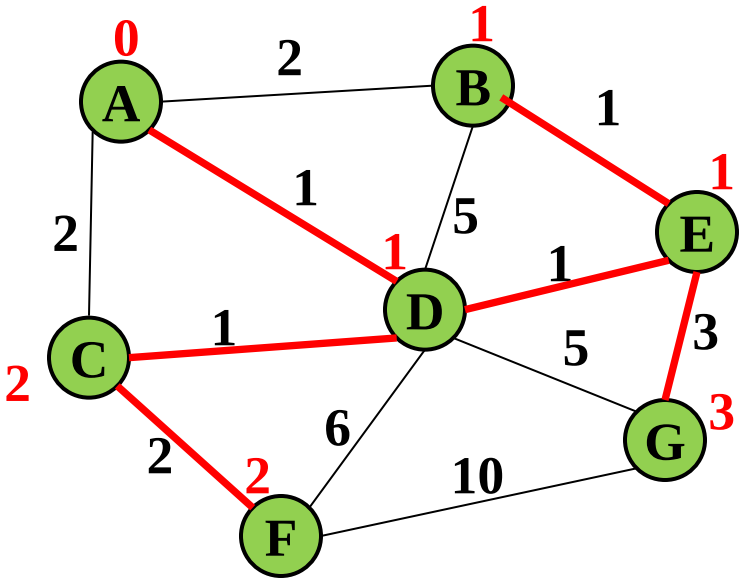
vertex	known?	cost	prev
A	Y	0	
B	Y	1	E
C	Y	1	D
D	Y	1	A
E	Y	1	D
F		2	C
G		3	E

Example



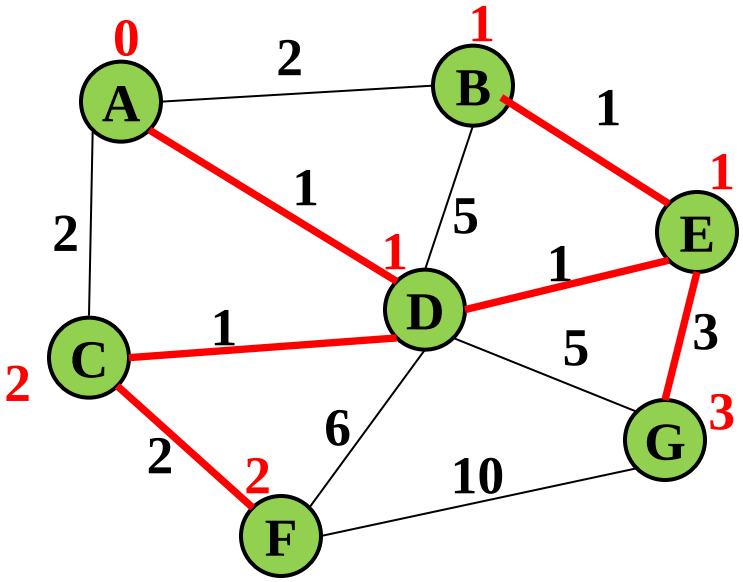
vertex	known?	cost	prev
A	Y	0	
B	Y	1	E
C	Y	1	D
D	Y	1	A
E	Y	1	D
F	Y	2	C
G		3	E

Example



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

Example



How would this differ for a directed graph?

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

Analysis

- Correctness ??
  - A bit tricky
  - Intuitively similar to Dijkstra
  - Won't have time to cover it this quarter

- Run-time
  - Same as Dijkstra
  - $O(|E| \log |V|)$ using a priority queue

Aside: Cycle detection

- For Kruskal's, we'll need efficient detection of cycles in a graph
- To decide if an edge could form a cycle, it takes $O(|E|)$ time since we may need to do a full traversal

- But there is a faster way using the disjoint-set ADT
  - Initially, each item is in its own 1-element set
  - **`find(u,v)`**: are **`u`** and **`v`** in the same set?
  - **`union(u,v)`**: union (combine) the sets **`u`** and **`v`** are in
  - Result: We can check whether adding an edge will result in a cycle in O(log|V|) time

  Check out the textbook for details

  Was covered in 326; didn't make it to 332

Kruskal's Algorithm

Idea: Grow a forest out of edges that do not grow a cycle

So:

- Sort edges: $O(|E| \log |E|)$
- Iterate through edges using union-find for cycle detection $O(|E| \log |V|)$

Somewhat better:

- Floyd's algorithm to build min-heap with edges $O(|E|)$
- Iterate through edges using union-find for cycle detection and `deleteMin` to get next edge $O(|E| \log |V|)$
- (Not better worst-case asymptotically, but often stop long before considering all edges)

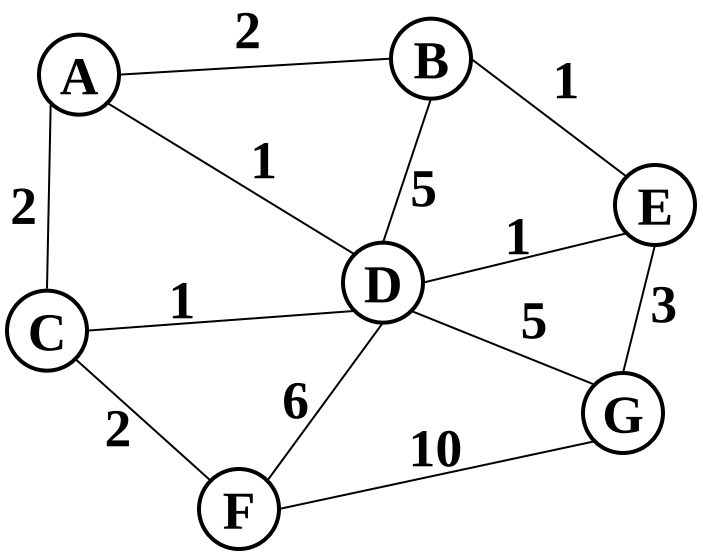
Kruskal's Algorithm: Pseudocode

1. Sort edges by weight (or put in min-heap)
2. Each node in its own set
3. While output size **< |V|-1**
   - Consider next smallest edge **`(u,v)`**
   - if **`find(u,v)`** indicates **`u`** and **`v`** are in different sets, we know adding the edge won't form a cycle
     - Add the edge: output **`(u,v)`**
     - **`union(u,v)`**

Loop invariant:

`u` and **`v`** in same set if and only if connected in output-so-far

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)

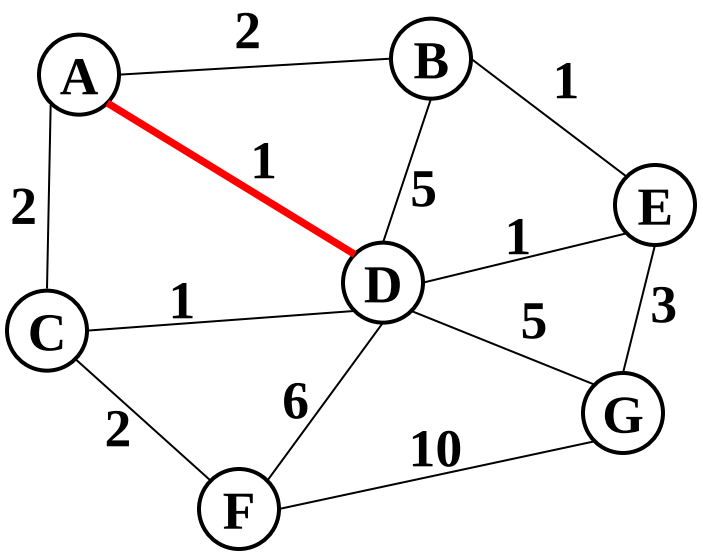
6: (D,F)

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)

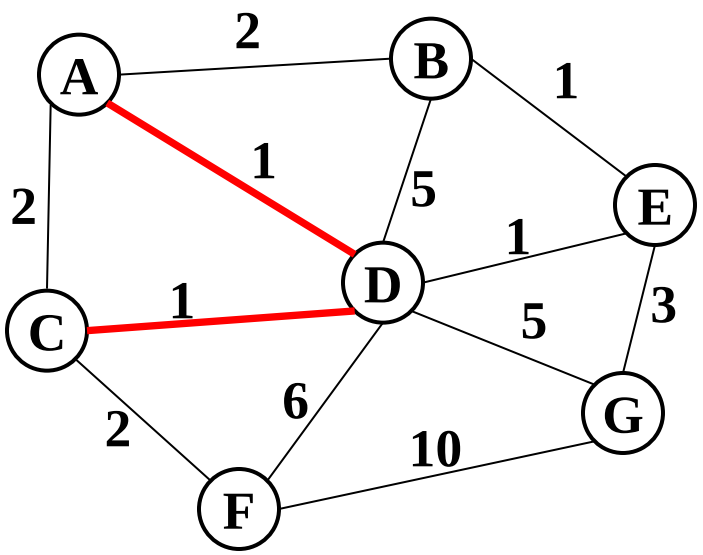
6: (D,F)

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)

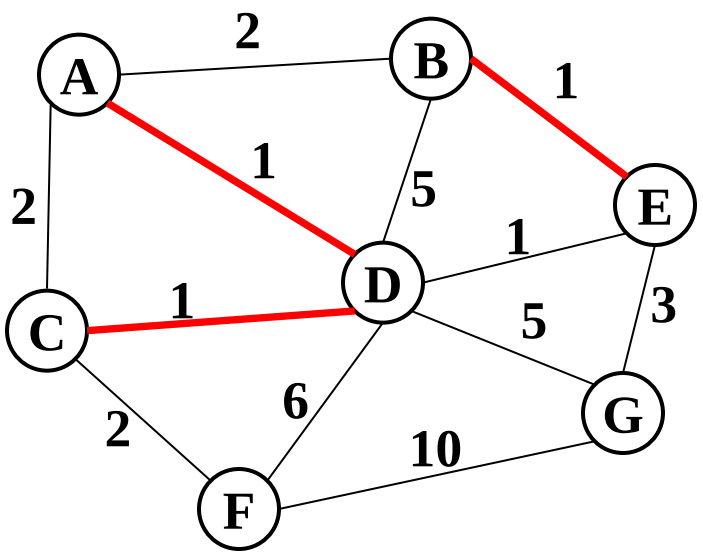
6: (D,F)

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)

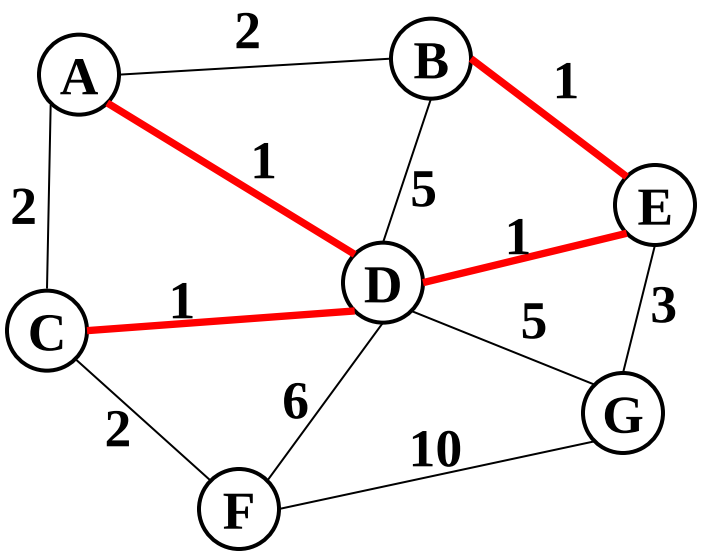
6: (D,F)

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)

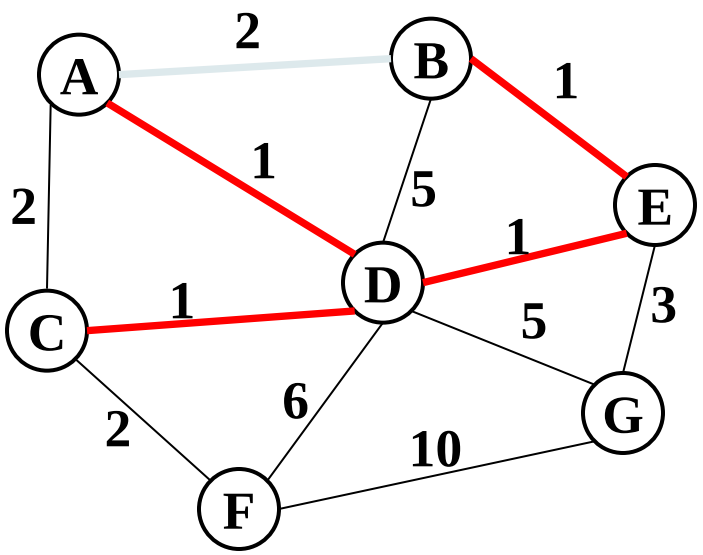
6: (D,F)

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)

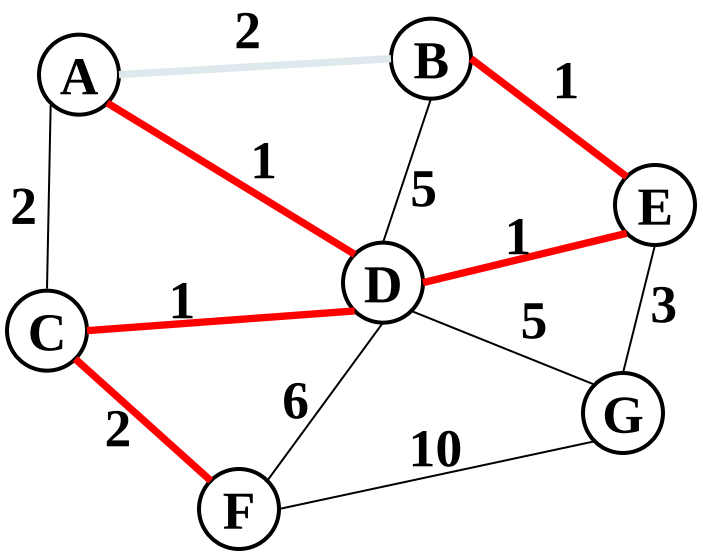
6: (D,F)

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)

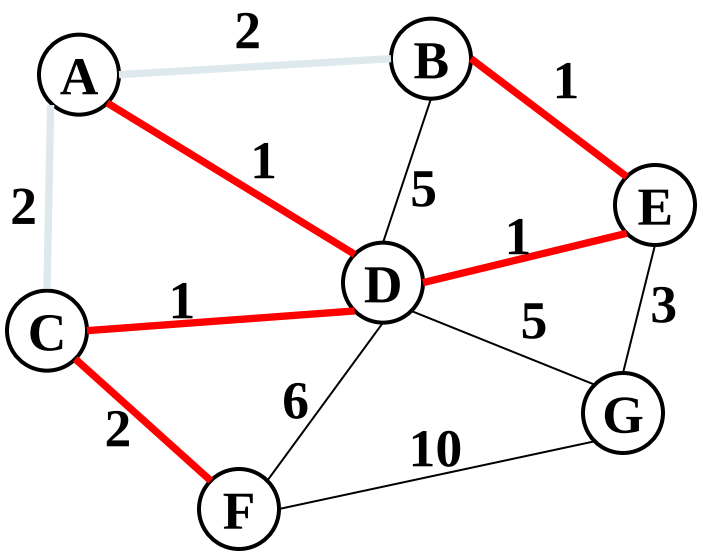
6: (D,F)

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)

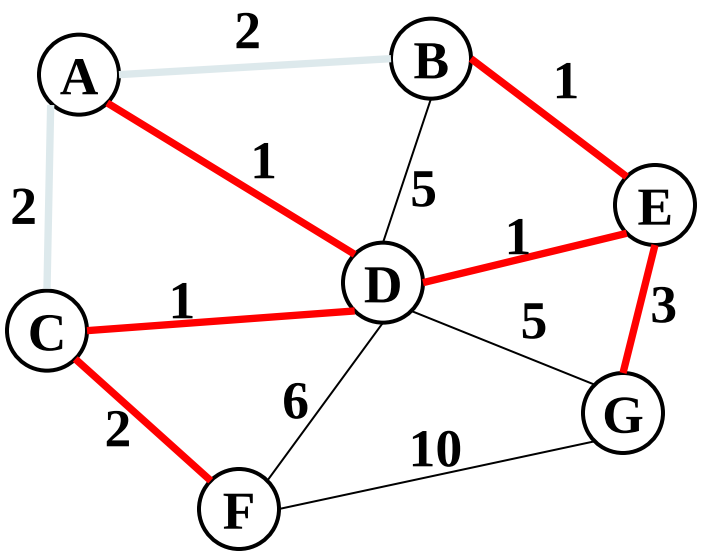
6: (D,F)

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

Correctness

Kruskal's algorithm is clever, simple, and efficient

- But does it generate a minimum spanning tree?
- How can we prove it?

First: it generates a spanning tree

- Intuition: Graph started connected and we added every edge that did not create a cycle
- Proof by contradiction: Suppose `u` and `v` are disconnected in Kruskal's result. Then there's a path from `u` to `v` in the initial graph with an edge we could add without creating a cycle. But Kruskal would have added that edge. Contradiction.

Second: There is no spanning tree with lower total cost

Won't cover the proof here

If you're interested, check it out on Wikipedia