CSE 332: Data Structures & Parallelism

Lecture 24: Minimum Spanning Trees

Ruth Anderson

Autumn 2025

Administrative

- EX09 – On Fork Join, Due TONIGHT, Fri Nov 21
- EX10 – Concurrency, Due **Mon** Nov 24
- EX11 – MSTs, programming, coming soon!
- Resources!
  - **Conceptual Office Hours**: 11:30 Tues (Connor) and 11:30 Wed (Samarth) both in CSE1 006. A space to ask about **course content and topics only** *as opposed to direct help with exercises*.
  - 1-on-1 Meeting Requests - Request a meeting with a staff member if you cannot make it to regularly scheduled office hours, or feel like you have an issue that requires a more in depth discussion.

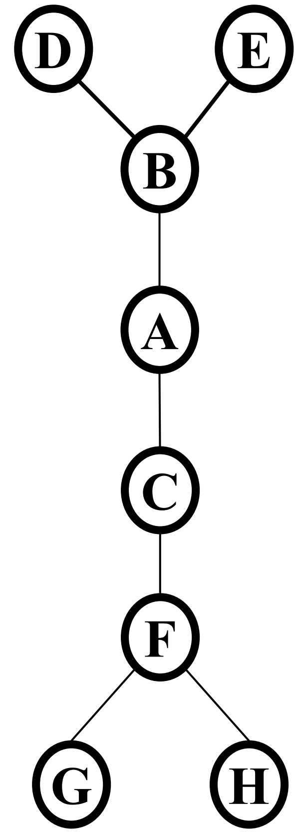
Trees as graphs

When talking about graphs,

we say a tree is a graph that is:

- undirected
- acyclic
- connected

Example:

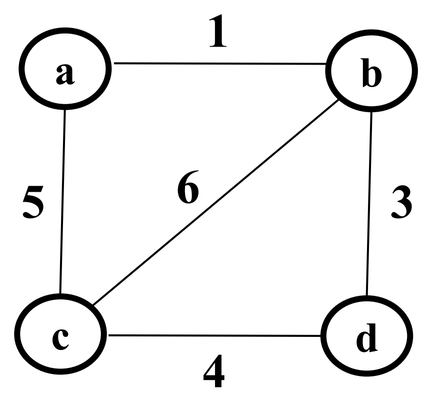


Minimum Spanning Trees

Given an undirected graph **G**=(**V**, **E**), find a graph **G'=(V, E')** such that:

- E' is a subset of E
- **G'** is connected
- G' has no cycles
- |E'| = |V| - 1
- $\sum_{(u,v)\in E'} c_{uv}$ is minimal

G' is a minimum spanning tree.

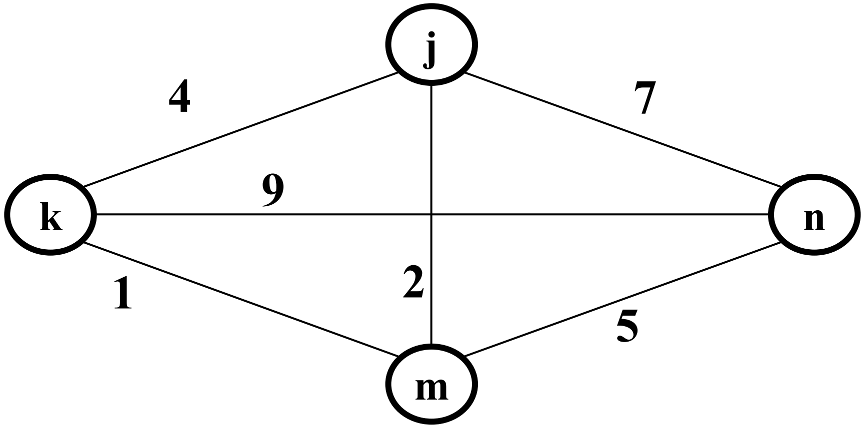


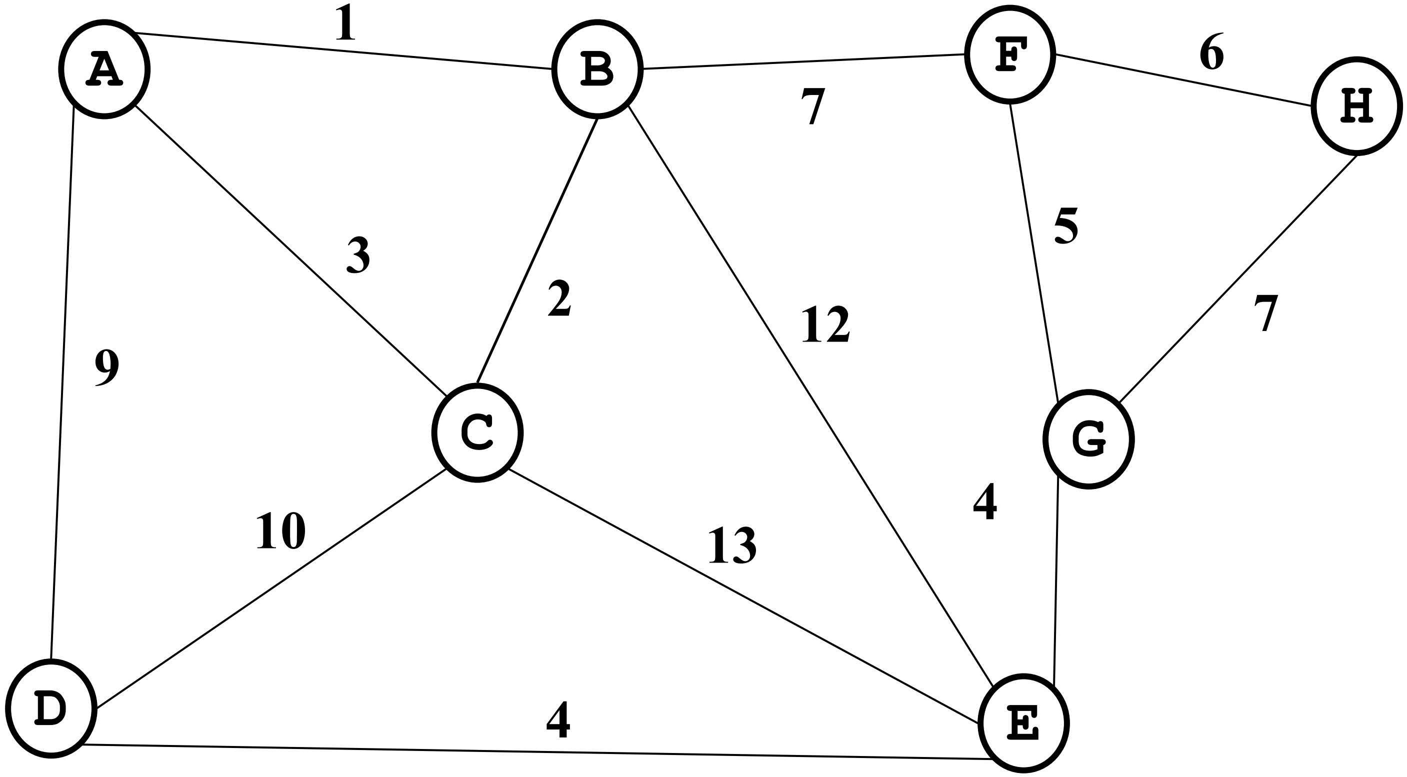
Applications:

- Example: Electrical wiring for a house or clock wires on a chip
- Example: A road network if you cared about asphalt cost rather than travel time

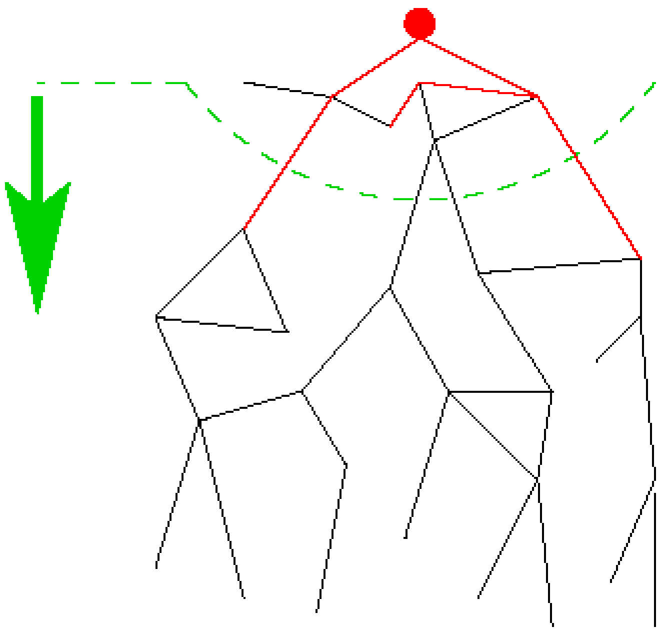
Student Activity

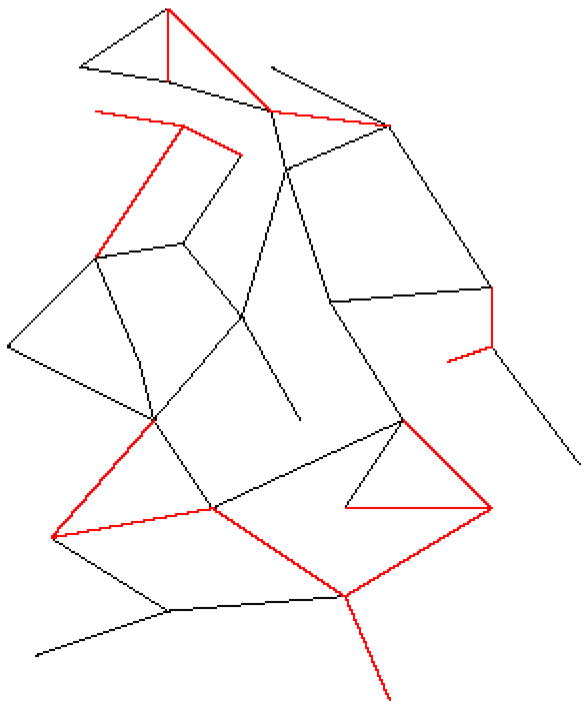
Find a MST





Two Different Approaches





Prim's Algorithm

Almost identical to Dijkstra's

Kruskals's Algorithm

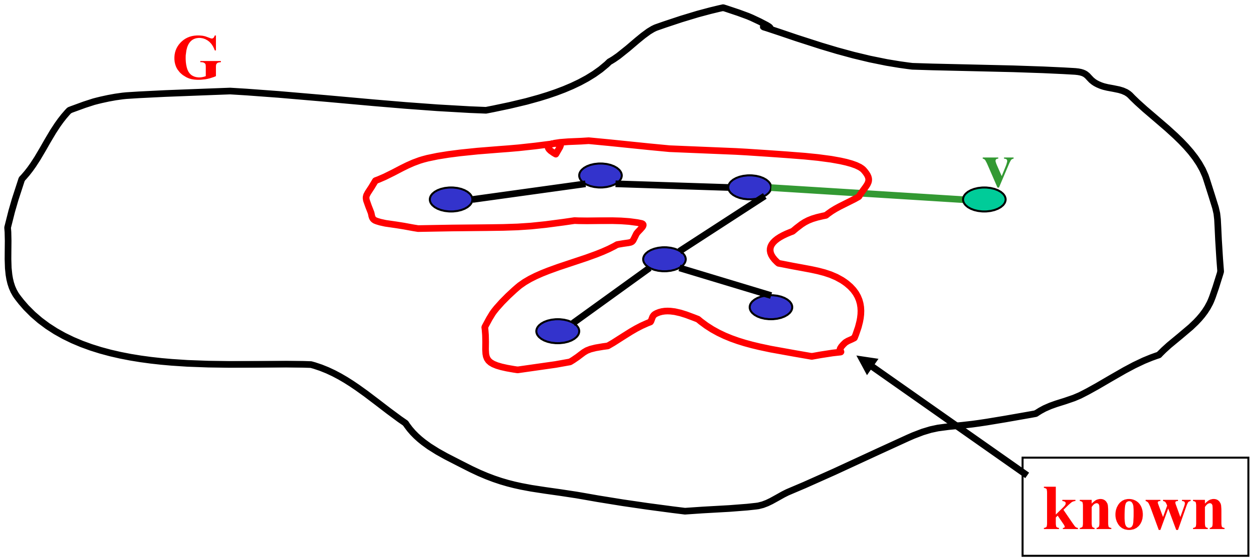
Completely different!

Prim's algorithm

Idea: Grow a tree by picking a vertex from the unknown set that has the smallest cost. Here cost = cost of the edge that connects that vertex to the known set. *Pick the vertex with the smallest cost that connects "known" to "unknown."*

A ***node-based*** greedy algorithm

Builds MST by greedily adding nodes



Prim's Algorithm vs. Dijkstra's

Recall:

Dijkstra picked the unknown vertex with smallest cost where
cost = ***distance to the source***.

Prim's pick the unknown vertex with smallest cost where
cost = ***distance from this vertex to the known set*** (in other words, the cost of the smallest edge connecting this vertex to the known set)

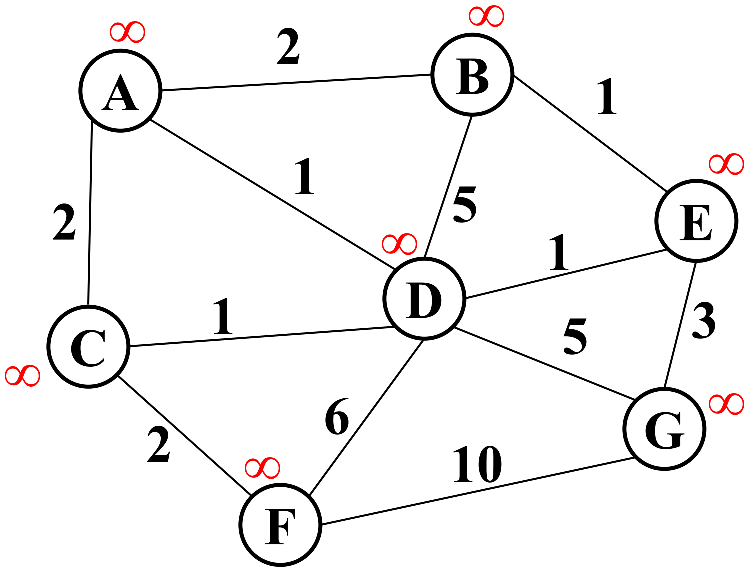
- Otherwise identical
- Compare to slides in Dijkstra lecture!

Prim's Algorithm for MST

1. For each node **`v`**, set **`v.cost = ∞`** and **`v.known = false`**
2. Choose any node **`v.`** (this is like your "start" vertex in Dijkstra)
   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 (the MST)
   c) For each edge **`(v,u)`** with weight **`w`**, where **`u`** is unknown:

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

Example: Find MST using Prim's



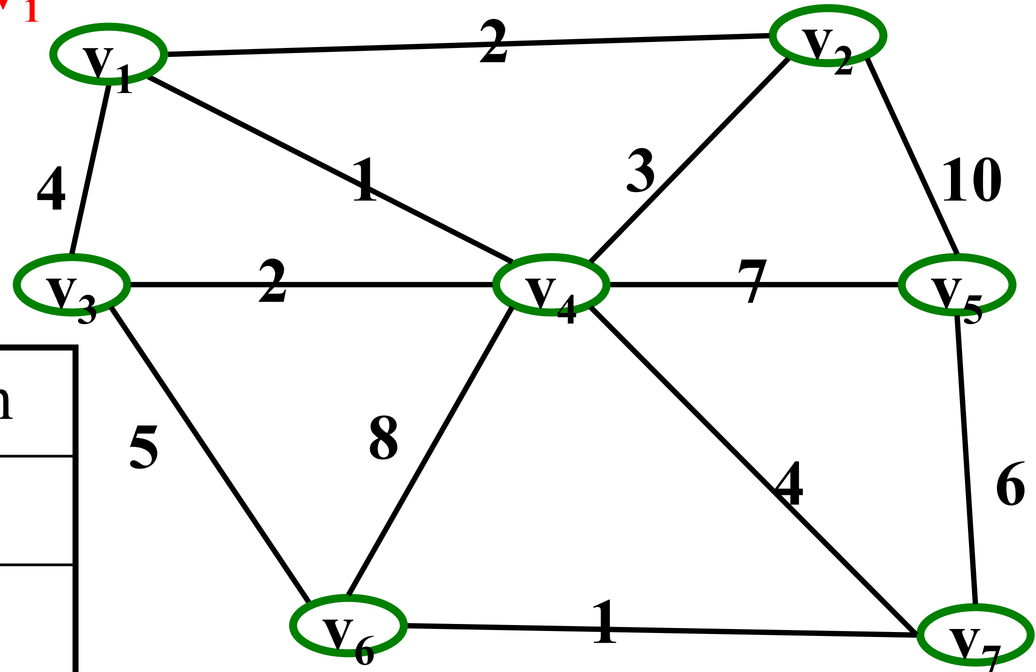
Order added to known set:

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

Student Activity

Start with V_1

Find MST using Prim's



V	Kwn	Distance	path
v1			
v2			
v3			
v4			
v5			
v6			
v7			

Order Declared Known:

V_1

Total Cost:

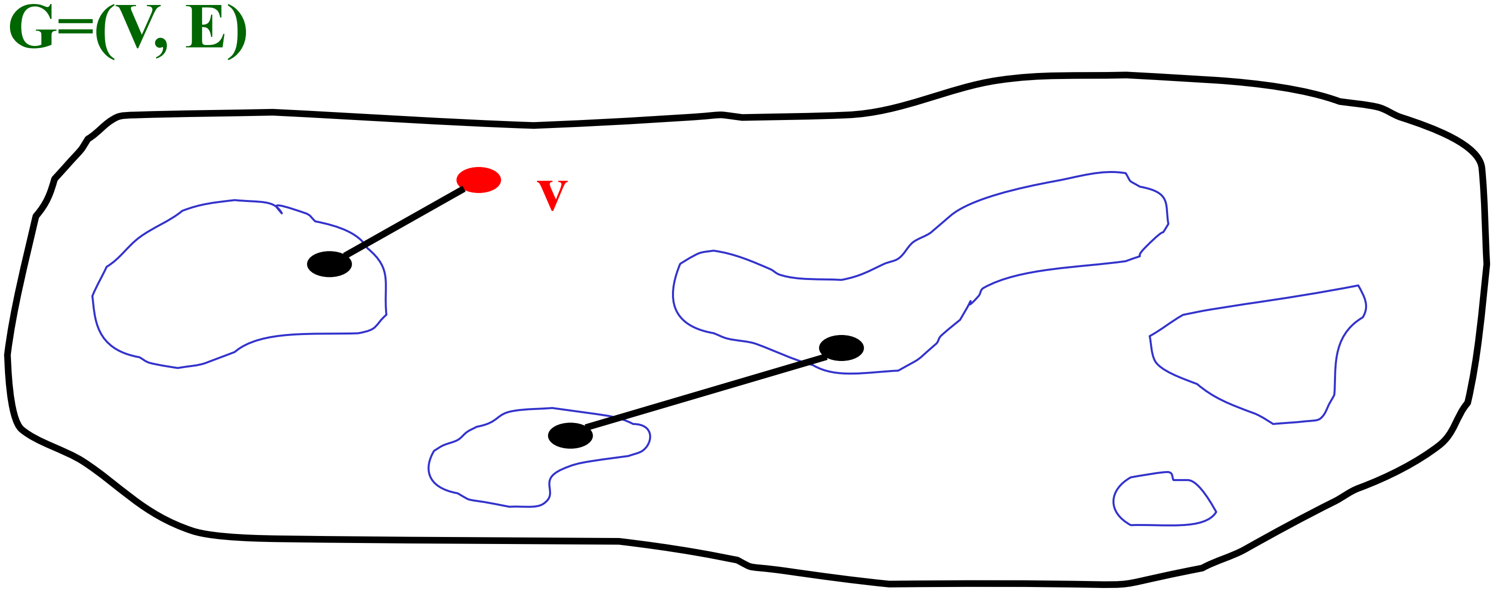
Prim's Analysis

- Correctness
  - Intuitively similar to Dijkstra

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

Kruskal's MST Algorithm

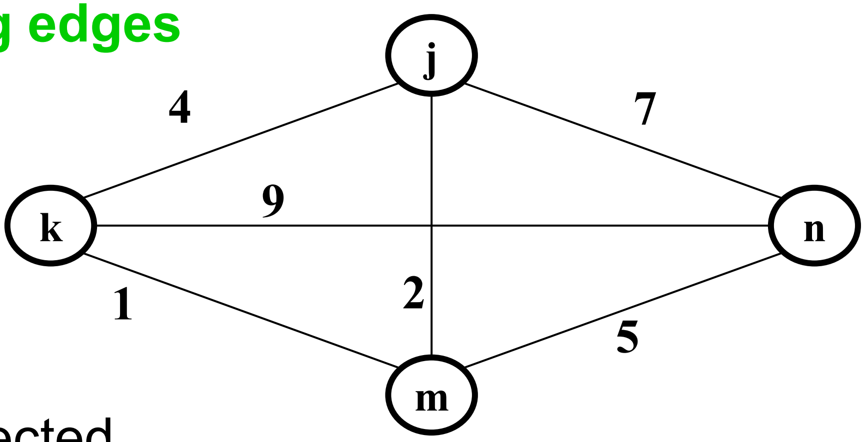
Idea: Grow a forest out of edges that do not create a cycle. Pick an edge with the smallest weight.



Kruskal's Algorithm for MST

An *edge-based* greedy algorithm

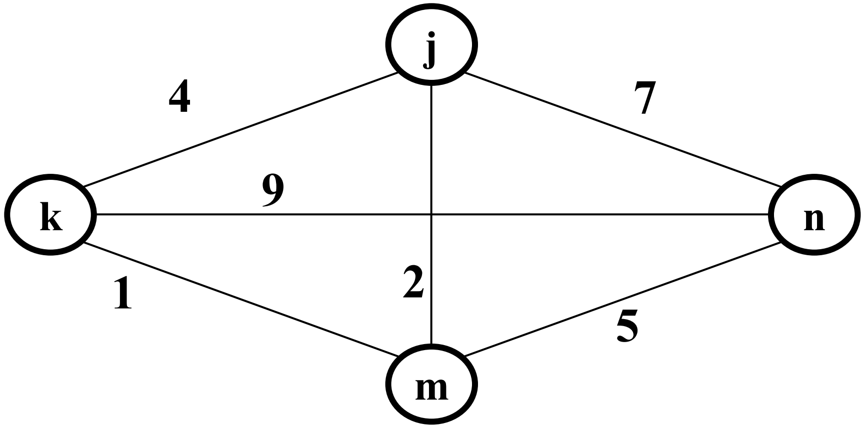
Builds MST by greedily adding edges

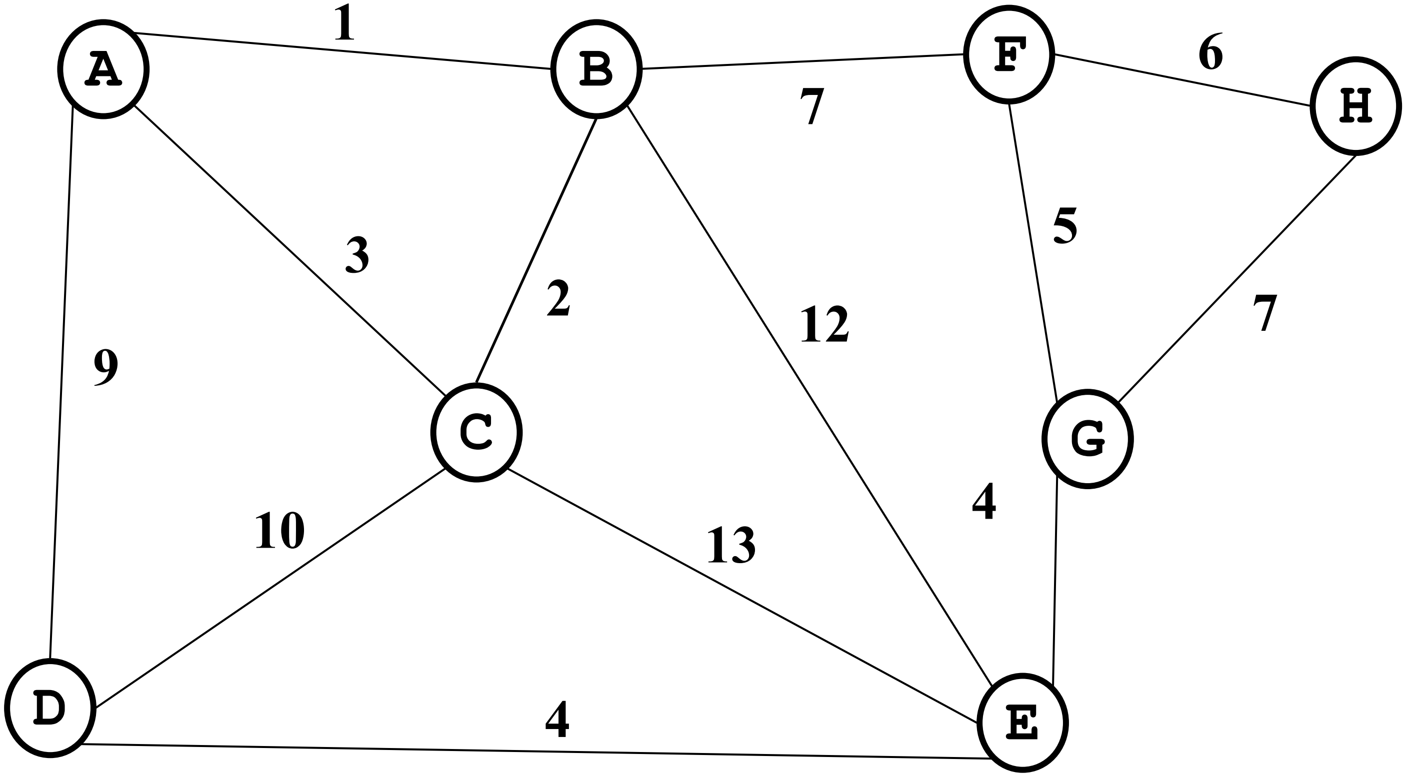


1. Initialize with
   - empty MST
   - all vertices marked unconnected
   - all edges unmarked
2. While all vertices are not connected
   a. Pick the <u>lowest cost edge</u> `(u,v)` and mark it
   b. If `u` and `v` are not already connected, add `(u,v)` to the MST and mark `u` and `v` as connected to each other

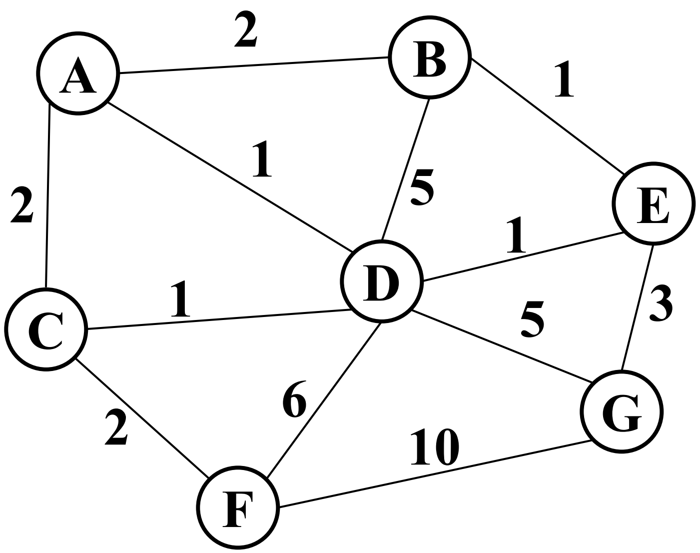
Student Activity

Find a MST





Example: Find MST using Kruskal's



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

Aside: Union-Find aka Disjoint Set ADT

- **Union(x,y)** – take the union of two sets named x and y
  - Given sets: {3,5,7} , {4,2,8}, {9}, {1,6}
  - **Union(5,1)**

    Result: {3,5,7,1,6}, {4,2,8}, {9},

  To perform the union operation, we replace sets x and y by $(x \cup y)$

- **Find(x)** – return the name of the set containing x.
  - Given sets: {3,5,7,1,6}, {4,2,8}, {9},
  - **Find(1)** returns 5
  - **Find(4)** returns 8

- We can do **Union** in constant time.
- We can get **Find** to be ***amortized*** constant time

  (worst case O(log n) for an individual **Find** operation).

Kruskal's pseudo code

```
void Graph::kruskal(){
  int edgesAccepted = 0;
  DisjSet s(NUM_VERTICES);

  while (edgesAccepted < NUM_VERTICES – 1){
    e = smallest weight edge not deleted yet;
    // edge e = (u, v)
    uset = s.find(u);
    vset = s.find(v);
    if (uset != vset){
      edgesAccepted++;
      s.unionSets(uset, vset);
    }
  }
}
```