CS 225

Data Structures

April 23 – MST II

Brad Solomon

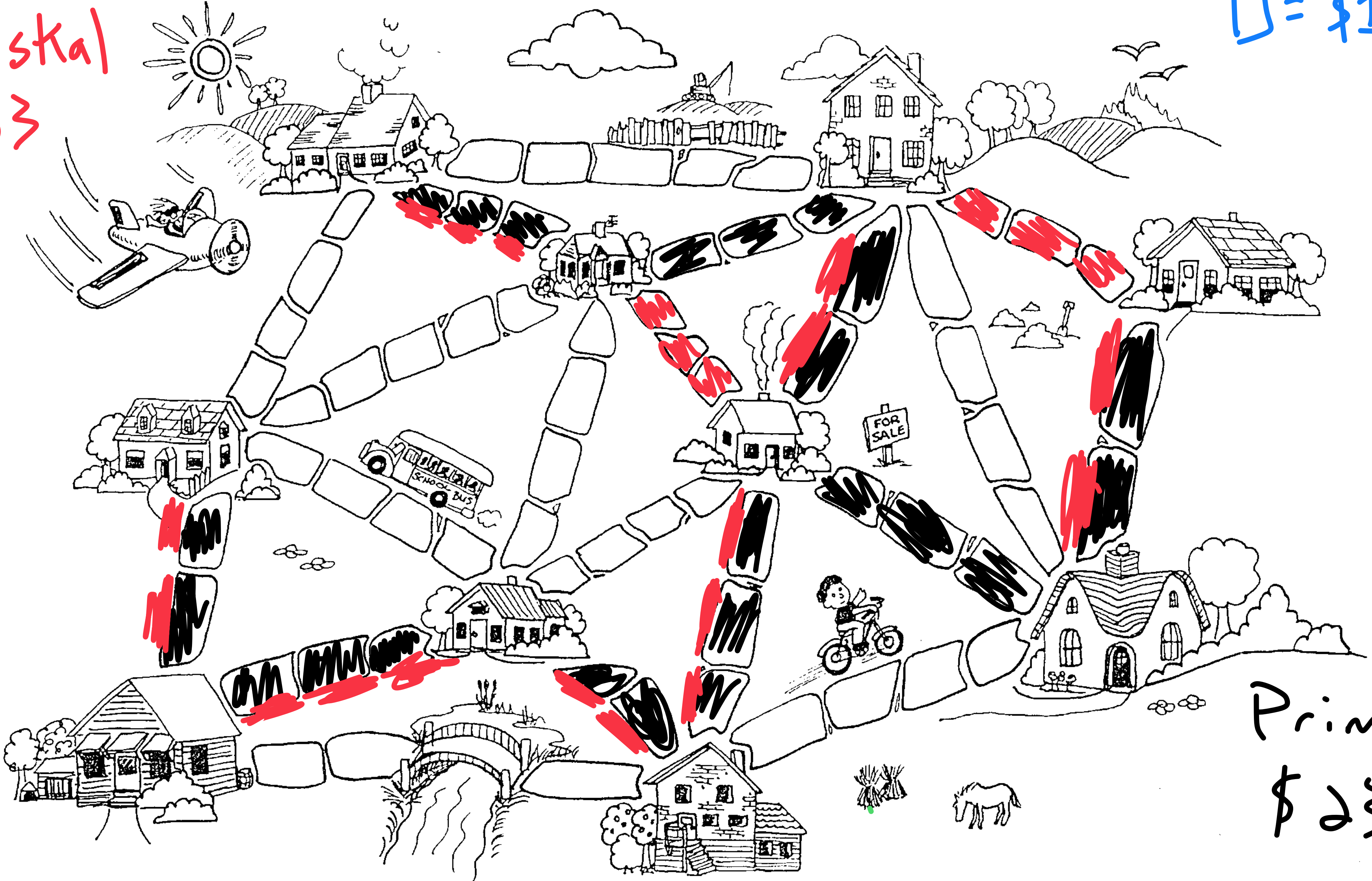
Learning Objectives

- Formalize Minimum Spanning Tree (MST)
- Analyze Kruskal and Prims' respective algorithms
- Compare runtimes and implementation strategies

Kruskal

$23

□ = $1



Prim

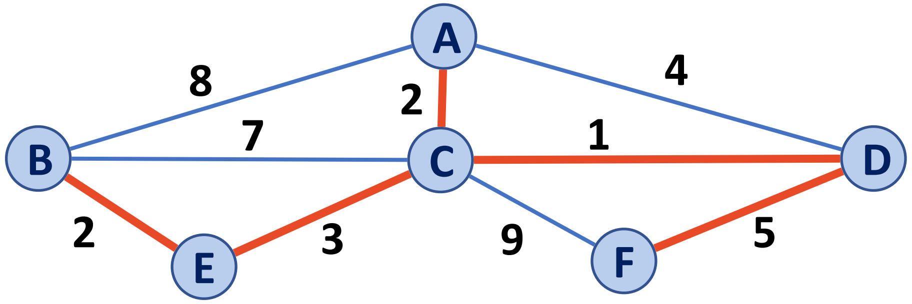
$23

Minimum Spanning Tree Algorithms

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

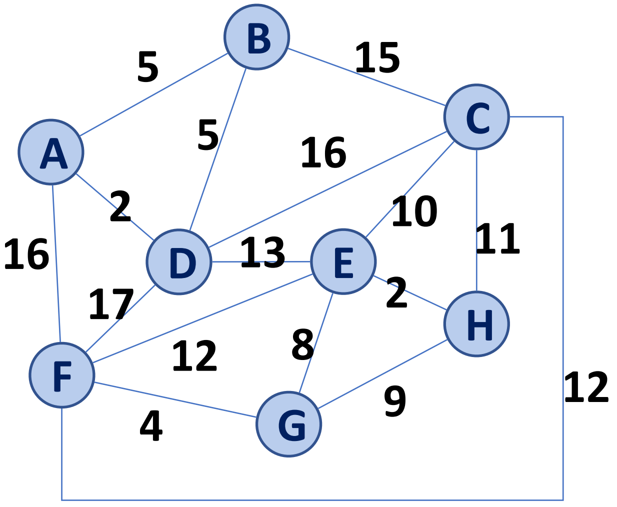
Output: A graph G' with the following properties:

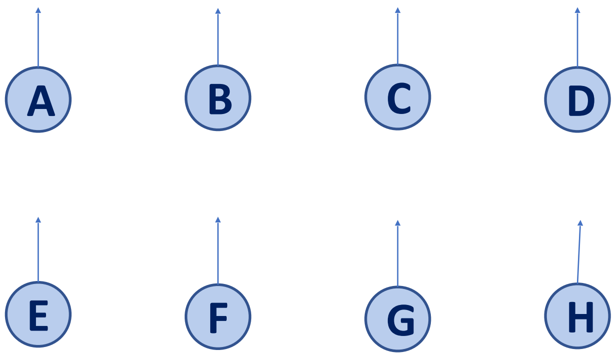
- G' is a spanning graph of G
- G' is a tree (connected, acyclic)
- G' has a minimal total weight among all spanning trees



Kruskal's Algorithm

- (A, 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):
    forest.makeSet(v)

  PriorityQueue Q    // min edge weight
  foreach (Edge e : G):
    Q.insert(e)

  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:	Heap	Sorted Array
Building (Line 6-8)		
Each removeMin (Line 13)		

```
1  KruskalMST(G):
2    DisjointSets forest
3    foreach (Vertex v : G):
4      forest.makeSet(v)
5
6    PriorityQueue Q    // min edge weight
7    foreach (Edge e : G):
8      Q.insert(e)
9
10   Graph T = (V, {})
11
12   while |T.edges()| < n-1:
13     Vertex (u, v) = Q.removeMin()
14     if forest.find(u) != forest.find(v):
15        T.addEdge(u, v)
16        forest.union( forest.find(u),
17                      forest.find(v) )
18
19   return T
```

Kruskal's Algorithm

Priority Queue:	Total Running Time
Heap	
Sorted Array	

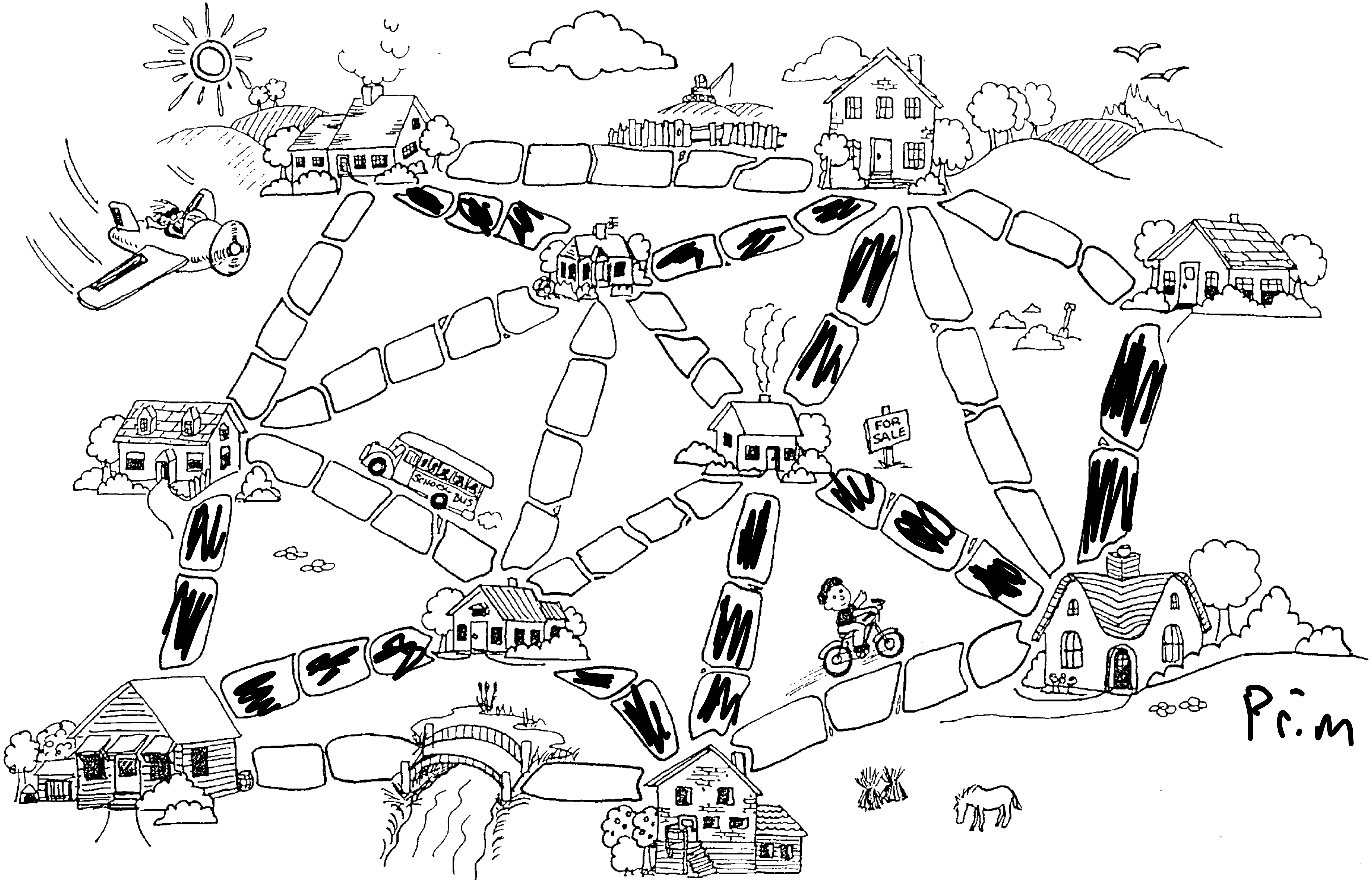
```
KruskalMST(G):
  DisjointSets forest
  foreach (Vertex v : G):
    forest.makeSet(v)

  PriorityQueue Q     // min edge weight
  foreach (Edge e : G):
    Q.insert(e)

  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

Which Priority Queue Implementation is better for running Kruskal’s Algorithm?

- Heap:

- Sorted Array:

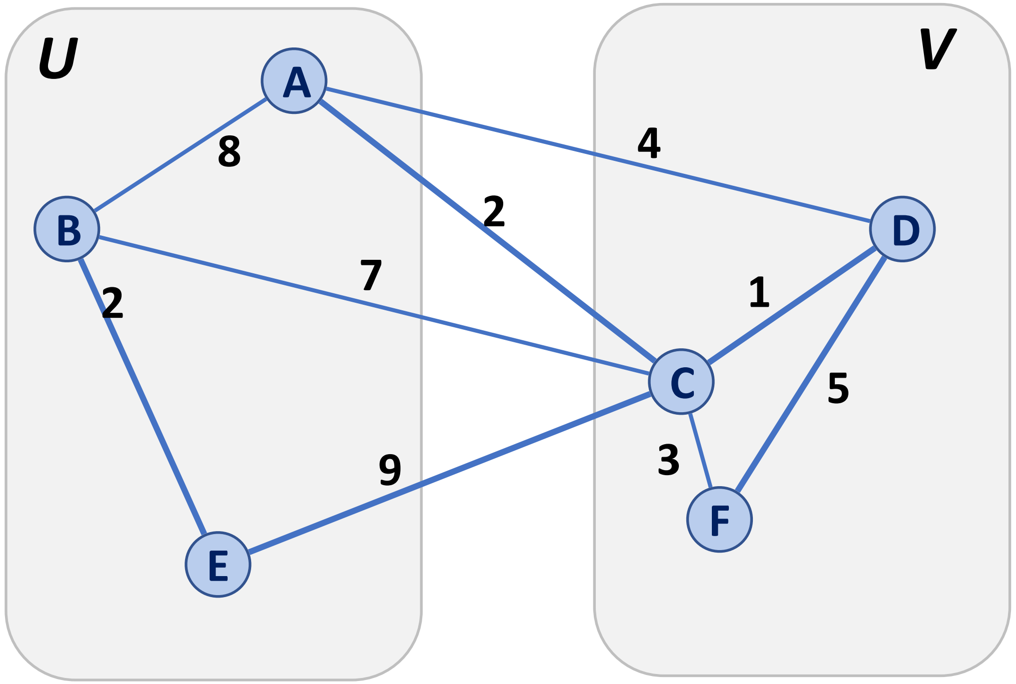

SCHOOL BUS
FOR SALE
Prim

Partition Property

Consider an arbitrary partition of the vertices on **G** into two subsets **U** and **V**.

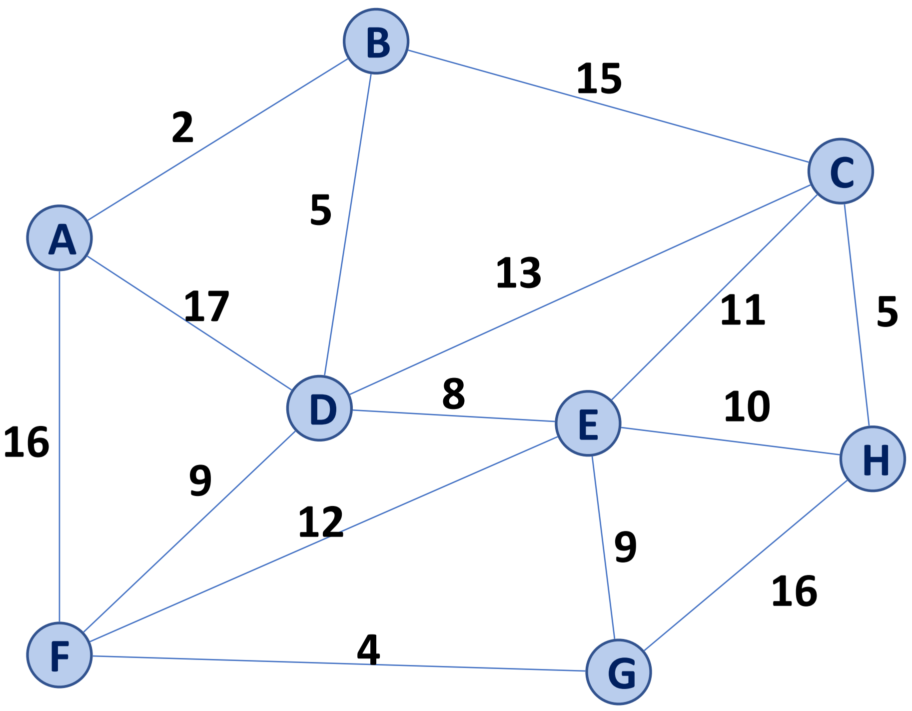
Let **e** be an edge of minimum weight across the partition.

Then **e** is part of some minimum spanning tree.

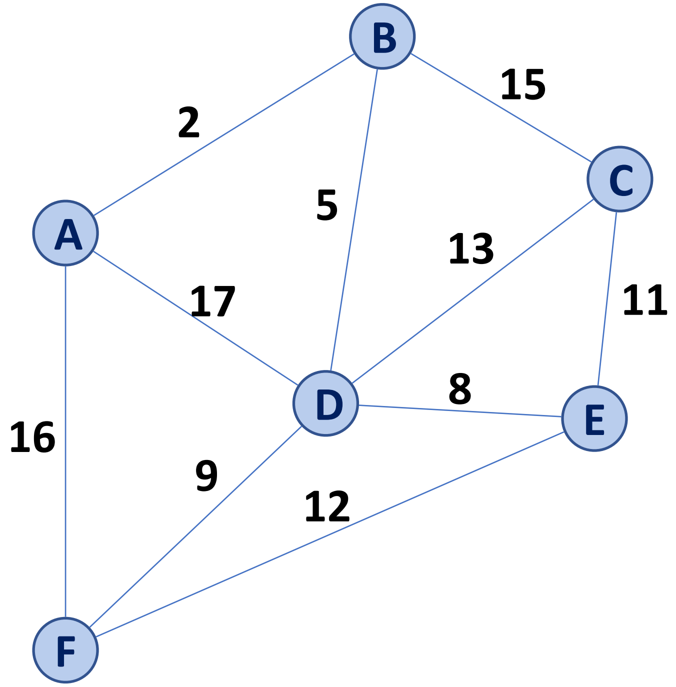


Partition Property

The partition property suggests an algorithm:



Prim's Algorithm




```
PrimMST(G, s):
  Input: G, Graph;
         s, vertex in G, starting vertex
  Output: T, a minimum spanning tree (MST) of G

  foreach (Vertex v : G):
    d[v] = +inf
    p[v] = NULL
  d[s] = 0

  PriorityQueue Q   // min distance, defined by d[v]
  Q.buildHeap(G.vertices())
  Graph T           // "labeled set"

  repeat n times:
    Vertex u = Q.removeMin()
    T.add(u)
    foreach (Vertex v : neighbors of u not in T):
      if cost(v, u) < d[v]:
        d[v] = cost(v, u)
        p[v] = u

  return T
```

Prim's Algorithm

```
PrimMST(G, s):
  foreach (Vertex v : G):
    d[v] = +inf
    p[v] = NULL
  d[s] = 0

  PriorityQueue Q // min distance, defined by d[v]
  Q.buildHeap(G.vertices())
  Graph T          // "labeled set"

  repeat n times:
    Vertex u = Q.removeMin()
    T.add(u)
    foreach (Vertex v : neighbors of u not in T):
      if cost(v, u) < d[v]:
        d[v] = cost(v, u)
        p[v] = u
```

	Adj. Matrix	Adj. List
Heap		
Unsorted Array		

Prim's Algorithm

```
PrimMST(G, s):
  foreach (Vertex v : G):
    d[v] = +inf
    p[v] = NULL
  d[s] = 0

  PriorityQueue Q // min distance, defined by d[v]
  Q.buildHeap(G.vertices())
  Graph T         // "labeled set"

  repeat n times:
    Vertex u = Q.removeMin()
    T.add(u)
    foreach (Vertex v : neighbors of u not in T):
      if cost(v, u) < d[v]:
        d[v] = cost(v, u)
        p[v] = u
```

	Adj. Matrix	Adj. List
Heap	$O(n^2 + m \lg(n))$	$O(n \lg(n) + m \lg(n))$
Unsorted Array	$O(n^2)$	$O(n^2)$

MST Algorithm Runtime:

- Kruskal's Algorithm:

  O(n + m lg(n))

- Prim's Algorithm:

  O(n lg(n) + m lg(n))

- What must be true about the connectivity of a graph when running an MST algorithm?

  - How does n and m relate?

MST Algorithm Runtime:

- Kruskal's Algorithm:

  $O(n + m \lg(n))$

- Prim's Algorithm:

  $O(n \lg(n) + m \lg(n))$

Sparse Graph:

Dense Graph:

Suppose I have a new heap:

	Binary Heap	Fibonacci Heap
Remove Min	O(lg(n))	O(lg(n))
Decrease Key	O(lg(n))	O(1)*

What's the updated running time?

```
PrimMST(G, s):
  foreach (Vertex v : G):
    d[v] = +inf
    p[v] = NULL
  d[s] = 0

  PriorityQueue Q // min distance, defined by d[v]
  Q.buildHeap(G.vertices())
  Graph T          // "labeled set"

  repeat n times:
    Vertex m = Q.removeMin()
    T.add(m)
    foreach (Vertex v : neighbors of m not in T):
      if cost(v, m) < d[v]:
        d[v] = cost(v, m)
        p[v] = m
```

Final Big-O MST Algorithm Runtimes:

- Kruskal's Algorithm:
  O(m lg(n))
- Prim's Algorithm:
  O(n lg(n) + m)

Sparse Graph:

Dense Graph: